

CHAPTER 1: DEFI FUNDAMENTALS

Decentralized Finance reimagines every financial service without intermediaries. No banks approving your loans. No brokers executing your trades. No governments freezing your accounts. Just code running on blockchains, accessible to anyone with an internet connection. This chapter establishes the conceptual foundation for understanding DeFi before diving into specific protocols.

SECTION 1: WHAT IS DEFI

The Core Concept

DeFi stands for Decentralized Finance. It refers to financial services built on public blockchains using smart contracts instead of traditional institutions. Where traditional finance requires trusted intermediaries, DeFi replaces them with transparent, auditable code.

A bank holds your deposits and decides whether to approve your loan application. A DeFi lending protocol holds deposits in a smart contract and algorithmically determines lending terms based on supply and demand. Both serve similar functions. The difference is who controls the process.

Traditional financial institutions are black boxes. You deposit money and trust they will return it. You apply for a loan and hope they approve it. Their internal logic, risk models, and decision criteria remain hidden. DeFi protocols are open source. Anyone can read the code, verify the logic, and understand exactly how their funds are managed.

The Building Blocks

DeFi applications compose together like Lego blocks. Each protocol provides a specific function that others can build upon.

Stablecoins provide price-stable assets for trading and lending without cryptocurrency volatility.

Decentralized exchanges enable token swaps without centralized order books or custody.

Lending protocols let users borrow against collateral or earn interest on deposits.

Yield aggregators automatically move funds between protocols to maximize returns.

Derivatives platforms offer options, futures, and synthetic assets.

Insurance protocols provide coverage against smart contract failures.

These components interact. You might deposit ETH into a lending protocol, borrow stablecoins against it, swap those stablecoins for another token on a DEX, and stake that token to earn yield. Each step uses a different protocol, but they compose seamlessly because they all run on the same blockchain.

Permissionless Access

Traditional finance requires permission. Opening a bank account requires identity verification. Getting a loan requires credit checks. Trading stocks requires brokerage approval. These barriers exclude billions of people worldwide.

DeFi requires only a wallet. No identity verification. No credit history. No minimum balance. If you can access the internet and pay gas fees, you can use DeFi. A farmer in rural Africa has the same access as a Wall Street trader.

This permissionless nature is both a feature and a challenge. It enables financial inclusion for the unbanked. It also enables money laundering, tax evasion, and sanctions circumvention. Regulators worldwide grapple with how to preserve benefits while preventing abuse.

Transparency and Auditability

Every DeFi transaction is recorded on a public blockchain. Anyone can verify balances, track fund flows, and audit protocol behavior. This transparency creates accountability impossible in traditional finance.

When a bank fails, depositors often learn only after their money is gone. When a DeFi protocol has issues, on-chain observers can see warning signs in real time. Large withdrawals, unusual transactions, and declining reserves are all publicly visible.

Protocol code is typically open source. Security researchers, competitors, and users can all review the logic. Bugs still occur, but they are found faster when millions of dollars incentivize discovery.

Composability

DeFi protocols are designed to interact. This composability means any protocol can build on any other without permission or integration agreements. Developers call this "money Legos" because protocols snap together like building

blocks.

A flash loan demonstrates extreme composability. In a single transaction you can borrow millions of dollars, swap tokens across multiple exchanges, repay the loan with profits, and keep the difference. If any step fails, the entire transaction reverts. No collateral required because the loan exists only within one atomic transaction.

Composability creates both innovation and risk. New protocols can leverage existing infrastructure. But dependencies create cascading failure potential. If a core protocol fails, everything built on it suffers.

SECTION 2: TVL AND DEFI METRICS

Total Value Locked

TVL (Total Value Locked) measures the dollar value of assets deposited in DeFi protocols. It is the most common metric for comparing protocol size and tracking ecosystem growth.

If a lending protocol holds 100,000 ETH and ETH trades at \$3,000, the protocol's TVL is \$300 million. If users deposit more ETH or ETH price rises, TVL increases. If users withdraw or prices fall, TVL decreases.

TVL has limitations. A protocol with \$1 billion TVL is not necessarily more valuable or successful than one with \$100 million. TVL says nothing about revenue, profitability, or actual usage. Some protocols incentivize deposits with token rewards, inflating TVL artificially.

Protocol Revenue

Revenue measures what users actually pay to use a protocol.

Swap fees on DEXs, interest spreads on lending protocols, and liquidation penalties all generate revenue.

Uniswap charges 0.3% on most swaps. If daily swap volume is \$1 billion, daily revenue is \$3 million. This revenue goes to liquidity providers (and sometimes token holders), not the Uniswap company.

Revenue is more meaningful than TVL for evaluating sustainability. A protocol can have high TVL but low revenue if users deposit but rarely transact. Conversely, a low-TVL protocol with high revenue indicates efficient capital utilization.

Fully Diluted Valuation

FDV (Fully Diluted Valuation) calculates total protocol value if all tokens were circulating. If a protocol has 1 billion total token supply and the token trades at \$5, FDV is \$5 billion, regardless of how many tokens actually circulate.

FDV matters because most protocols launch with limited circulating supply. Team tokens, investor allocations, and ecosystem reserves vest over years. Today's \$500 million market cap might represent only 10% of eventual supply, making true FDV \$5 billion.

Comparing market cap to FDV reveals inflation risk. A token with 10% circulating supply will face 90% potential dilution as remaining tokens unlock. Early investors often sell when their tokens vest, pressuring prices.

Price to Earnings Ratios

Traditional finance uses P/E ratios to compare valuations.

DeFi increasingly adopts similar metrics.

If a protocol has \$1 billion FDV and generates \$100 million annual revenue, its price-to-sales ratio is 10. This can be compared across protocols to identify relative over or under valuation.

The challenge is defining "earnings." Revenue goes to various parties: liquidity providers, token stakers, protocol treasuries, and development teams. Which flows count as earnings? Different analysts make different choices, making comparisons difficult.

Active Users and Transactions

User metrics measure actual adoption beyond capital deployment.

Daily Active Addresses counts unique wallet addresses interacting with a protocol each day. This undercounts humans (one person might use multiple wallets) and overcounts bots (one bot might use many addresses).

Transaction Count measures how often the protocol is used. High transaction counts with low TVL suggest active usage rather than passive deposits.

These metrics combined with TVL and revenue paint a fuller picture. A protocol with high TVL, high revenue, and high active users is likely healthy. High TVL but low revenue and few users suggests incentivized deposits that may leave when rewards end.

Analyzing Protocol Health

Combine multiple metrics for comprehensive analysis.

TVL trends show whether deposits are growing, stable, or declining. Sudden drops often indicate problems or lost confidence.

Revenue trends reveal whether usage is increasing. Growing revenue with stable TVL means improved capital efficiency.

Token distribution affects governance and price stability. Concentrated holdings mean few parties control the protocol.

Smart contract risk increases with code complexity and dependencies. More audits, longer deployment history, and simpler design reduce risk.

The following Python code fetches basic DeFi metrics from DeFiLlama API.

```
import requests

def get_protocol_tvl(protocol_slug):
    url = f"https://api.llama.fi/protocol/{protocol_slug}"
    response = requests.get(url)
    data = response.json()

    return {
        "name": data["name"],
        "tvl": data["tvl"],
        "chain_tvls": data.get("chainTvls", {}),
        "symbol": data.get("symbol"),
        "category": data.get("category")
    }

def get_top_protocols(limit = 10):
    url = "https://api.llama.fi/protocols"
    response = requests.get(url)
    data = response.json()
```

```
sorted_protocols = sorted(data, key=lambda x: x.get("tv1",
0), reverse=True)
```

```
return [{
    "name": p["name"],
    "tv1": p["tv1"],
    "category": p.get("category"),
    "chains": p.get("chains", [])
} for p in sorted_protocols[:limit]]
```

```
def get_chain_tv1(chain_name):
    url = f"https://api.llama.fi/v2/chains"
    response = requests.get(url)
    data = response.json()
```

```
for chain in data:
    if chain["name"].lower() == chain_name.lower():
        return {
            "name": chain["name"],
            "tv1": chain["tv1"],
            "tokenSymbol": chain.get("tokenSymbol")
        }
```

```
return None
```

SECTION 3: DEFI VS TRADITIONAL FINANCE

Speed and Settlement

Traditional securities trading settles in T + 2 (two business days). You buy stock Monday, it officially transfers Wednesday. During this time, counterparty risk exists. If the seller goes bankrupt before settlement, problems arise.

DeFi settles instantly. When you swap tokens on Uniswap,

ownership transfers in the same transaction. No waiting period. No counterparty risk. The trade either completes entirely or reverts entirely.

Cross-border payments illustrate this further. International wire transfers take 1-5 business days and cost \$25-50 in fees. Sending stablecoins anywhere in the world takes seconds and costs under \$1 on many chains.

Access and Hours

Traditional markets have hours. US stock markets operate 9:30 AM to 4:00 PM Eastern, Monday through Friday. Bond markets close at 5:00 PM. Banks have limited weekend hours or close entirely.

DeFi never closes. Trade any time, any day, any holiday. The blockchain runs 24/7/365. This continuous operation enables arbitrage, rapid response to news, and access regardless of timezone.

Custody and Control

Traditional finance requires trusting custodians. Your broker holds your stocks. Your bank holds your cash. You have claims on these institutions, not direct ownership of assets.

DeFi enables self-custody. Your tokens sit in your wallet, controlled by your private key. No institution can freeze your account or deny withdrawals. This sovereignty comes with responsibility. Lose your keys, lose your funds.

Some users prefer custodial services for convenience and security. Others demand self-custody for independence. DeFi supports both models.

Transparency vs Privacy

Traditional finance operates behind closed doors. Bank balance sheets are reported quarterly with limited detail. Trading algorithms are proprietary secrets. Decision criteria for loans, insurance, and other services are opaque.

DeFi code and state are public. Anyone can see exactly how much collateral backs loans, what positions exist, and how the protocol functions. This transparency enables verification but eliminates privacy.

On-chain activity is pseudonymous but traceable. Wallet addresses do not reveal identity directly, but transaction patterns, on-ramps, and other metadata can de-anonymize users. Privacy-preserving techniques exist but add complexity.

Efficiency and Costs

Traditional finance has enormous overhead. Physical branches, compliance departments, executive salaries, and shareholder returns all require revenue. This overhead translates to fees and spreads paid by users.

DeFi protocols run on code with minimal operational costs. A smart contract needs no employees, no office space, no marketing budget. This efficiency allows lower fees. DEX swap fees are typically 0.3% compared to 1-2% for foreign exchange at traditional banks.

However, blockchain fees add costs traditional finance lacks. Gas costs vary from pennies to hundreds of dollars depending on network congestion and complexity. Layer 2 solutions reduce these costs but add complexity.

Regulatory Clarity

Traditional finance operates within clear regulatory frameworks. Banks are licensed and supervised. Securities laws define what can be traded and how. Insurance is regulated by state commissions. Violations have known penalties.

DeFi exists in regulatory gray areas. Is a governance token a security? Is a lending protocol a bank? Is a DEX an exchange? Regulators worldwide are still deciding. This uncertainty creates risk for builders and users.

Some jurisdictions embrace DeFi innovation. Others ban it entirely. The regulatory landscape shifts constantly. Builders must navigate this uncertainty while users must understand they may lack legal protections available in traditional finance.

SECTION 4: RISKS IN DEFI

Smart Contract Risk

Code has bugs. Every DeFi protocol is a potential target for exploits. Hackers constantly probe for vulnerabilities. One bug can drain millions of dollars in seconds.

The history of DeFi is littered with hacks. The DAO hack in 2016 lost \$60 million and caused Ethereum to hard fork. Cream Finance lost \$130 million in 2021. Ronin Bridge lost \$625 million in 2022. Euler Finance lost \$197 million in 2023.

Mitigating smart contract risk requires multiple audits from reputable firms, formal verification where possible, bug bounty programs, time-tested code, and insurance coverage.

Even with all precautions, risk never reaches zero.

Oracle Manipulation

Many DeFi protocols need external price data. Lending protocols must know collateral values. Derivatives need asset prices. This data comes from oracles, systems that bring off-chain information on-chain.

If an attacker manipulates the oracle, they can exploit protocols relying on it. Flash loan attacks often involve temporarily manipulating prices on low-liquidity markets to exploit dependent protocols.

Chainlink dominates the oracle market because it aggregates prices from multiple sources, making manipulation expensive. Using time-weighted average prices (TWAPs) rather than spot prices also reduces manipulation risk.

Liquidation Risk

Borrowing against collateral creates liquidation risk. If collateral value drops below required thresholds, the protocol sells collateral to repay debt. This typically happens at a discount, resulting in losses for the borrower.

Liquidations can cascade. Large positions being liquidated dump tokens on the market, pushing prices lower, triggering more liquidations. These cascades amplified losses during market crashes in 2020, 2021, and 2022.

Users can mitigate by maintaining conservative collateral ratios, monitoring positions actively, and using stable collateral. Protocols can mitigate with gradual liquidation mechanisms and circuit breakers.

Impermanent Loss

Providing liquidity to DEXs exposes you to impermanent loss. When token prices diverge from when you deposited, you end up with less value than simply holding the tokens.

The math is unintuitive. If one token doubles in price while the other stays stable, liquidity providers lose about 5.7% compared to holding. If one token goes up 5x, the loss exceeds 25%.

This loss is "impermanent" because it reverses if prices return to original ratios. But if you withdraw at divergent prices, the loss becomes permanent. Trading fees can offset impermanent loss, but only if volume is sufficient.

Rug Pulls and Scams

Pseudonymous teams can deploy malicious protocols. They attract deposits with high yields, then drain funds and disappear. These "rug pulls" are common in DeFi, especially with new protocols.

Warning signs include anonymous teams, unaudited code, excessive yields without clear source, locked liquidity that is not actually locked, and aggressive marketing. Due diligence is essential. If yields seem too good to be true, they probably are.

Regulatory Risk

Governments can and do intervene in DeFi. The US Treasury sanctioned Tornado Cash in 2022, making it illegal for US persons to interact with the protocol. Developers have been arrested for building DeFi tools.

Regulatory crackdowns can render tokens worthless, prevent protocol access, and create legal liability for users. Geographic restrictions, KYC requirements, and outright bans all threaten DeFi's permissionless nature.

SECTION 5: OPPORTUNITIES IN DEFI

Yield Opportunities

DeFi offers yields unavailable in traditional finance. While banks pay 0-5% on savings, DeFi protocols can offer 10-100% or more. These yields come from trading fees, lending interest, liquidity incentives, and token rewards.

Sustainable yields typically range from 3-15% and come from economic activity: trading fees from actual swaps, interest from actual borrowers. Higher yields usually involve token incentives that dilute value, higher risk strategies, or unsustainable models.

Finding sustainable yield requires understanding where the yield originates. If you cannot identify who pays the yield, you might be the one paying it.

Arbitrage

Price differences between venues create arbitrage opportunities. The same token might trade at \$100 on one DEX and \$100.50 on another. Buying on the cheaper venue and selling on the expensive one captures the spread.

Simple arbitrage is highly competitive. Bots monitor every block and execute faster than humans can react. Profitable arbitrage requires speed, capital, and sophisticated strategies.

Cross-chain arbitrage has more opportunities because bridging adds latency and complexity. A token might trade cheaper on Polygon than Ethereum, with the bridge time creating windows for profit.

Providing Liquidity

DEXs need liquidity to function. Liquidity providers earn fees from every trade. Popular trading pairs can generate substantial fee income.

Concentrated liquidity on Uniswap V3 allows capital-efficient positions. By providing liquidity in a narrow price range, you earn more fees per dollar deployed. But if price moves outside your range, you earn nothing.

Successful liquidity provision requires understanding trading volumes, price volatility, and fee tiers. Automated tools help manage positions as prices move.

Building Protocols

The largest opportunity is building. DeFi needs better protocols, improved user experiences, and novel mechanisms. Successful protocol builders can capture enormous value.

Uniswap's UNI token reached \$40 billion fully diluted valuation. Aave, Compound, Curve, and others created billions in value. Even smaller protocols have made their builders wealthy.

Building requires deep technical knowledge, security expertise, and market understanding. The bar is high but the potential rewards match.

SECTION 6: THE DEFI ECOSYSTEM MAP

Layer 1 Blockchains

DeFi runs on smart contract platforms. Each blockchain has its own DeFi ecosystem with varying characteristics.

Ethereum hosts the largest and most mature DeFi ecosystem. Most innovation happens here first. But high gas costs during congestion push users to alternatives.

Solana offers very low fees and fast finality. Its DeFi ecosystem grew rapidly, suffered from network instability, and has stabilized. Major protocols include Raydium, Marinade, and Jupiter.

BNB Chain (formerly BSC) provides EVM compatibility with lower fees. It attracted users priced out of Ethereum but has centralization concerns with only 21 validators.

Avalanche, Polygon, Arbitrum, Optimism, and Base all host substantial DeFi activity with varying tradeoffs between decentralization, cost, and speed.

DEX Landscape

Uniswap dominates Ethereum DEX trading with over 60% market share. Its V3 concentrated liquidity model has been widely copied.

Curve specializes in stablecoin and pegged asset swaps with very low slippage. Its vote-escrow tokenomics (veTokenomics) model has been widely adopted.

Balancer offers flexible liquidity pools with custom token weightings. It powers many protocol-owned liquidity

implementations.

On Solana, Jupiter aggregates liquidity across all DEXs, becoming the default swap interface. Raydium provides concentrated liquidity and order book hybrid trading.

Lending Protocols

Aave leads decentralized lending with the most features and multi-chain presence. It supports multiple collateral types, variable and stable interest rates, flash loans, and credit delegation.

Compound pioneered the cToken model where deposits receive interest-bearing tokens. Its governance token COMP launched the "yield farming" phenomenon in 2020.

MakerDAO created DAI, the largest decentralized stablecoin. Users deposit collateral to mint DAI, creating a lending relationship with the protocol.

Morpho optimizes lending by matching borrowers with lenders peer-to-peer when possible, capturing better rates than pool-based lending.

Liquid Staking

Proof-of-stake networks require staking tokens to secure the network. Liquid staking protocols issue derivative tokens representing staked positions, allowing users to earn staking yield while maintaining liquidity.

Lido dominates Ethereum liquid staking with stETH. It controls about 30% of all staked ETH, raising centralization concerns.

Rocket Pool offers more decentralized staking through permissionless node operators. Its rETH token has lower market share but higher decentralization.

On Solana, Marinade (mSOL) and Jito (JitoSOL) provide liquid staking with MEV rewards.

Derivatives and Synthetics

Perpetual futures dominate crypto derivatives. Unlike traditional futures, perpetuals never expire and use funding rates to track spot prices.

dYdX built the leading decentralized perpetuals exchange, handling billions in daily volume. It migrated from Ethereum to its own Cosmos chain for better performance.

GMX on Arbitrum and Avalanche offers perpetuals with liquidity provided by its GLP token. Traders profit from GLP holder losses and vice versa.

Synthetix creates synthetic assets tracking real-world prices. Users can trade synthetic stocks, commodities, and forex without holding underlying assets.

Yield Aggregators

Yield aggregators automate strategy execution, finding and capturing the best yields across protocols.

Yearn Finance pioneered automated yield strategies. Users deposit tokens, Yearn's strategies move funds between protocols to maximize returns.

Convex Finance optimizes Curve yields by aggregating voting power. It became essential infrastructure for Curve liquidity

management.

Beefy Finance operates across many chains, offering auto-compounding vaults for various yield opportunities.

SECTION 7: GETTING STARTED WITH DEFI

Essential Tools

Wallet: MetaMask, Rabby, or Coinbase Wallet for browser interaction. Hardware wallet for storing significant value.

Block Explorer: Etherscan for Ethereum, chain-specific explorers for other networks. Essential for verifying transactions and understanding protocols.

Portfolio Tracker: DeBank, Zapper, or Zerion to view all positions across protocols and chains in one interface.

Analytics: DeFiLlama for TVL data, Dune Analytics for custom queries, CoinGecko or CoinMarketCap for prices.

First Steps

Start on Layer 2 networks like Arbitrum, Base, or Optimism. Fees are much lower, reducing the cost of learning mistakes.

Begin with simple swaps on Uniswap or a similar DEX. Understand gas costs, slippage, and transaction confirmation.

Try supplying liquidity with a small amount. Experience impermanent loss directly with minimal capital at risk.

Deposit to a lending protocol. Understand health factors, interest rates, and collateral requirements.

Each step builds knowledge. Mistakes with small amounts teach lessons that prevent larger losses later.

Risk Management

Never invest more than you can afford to lose entirely. DeFi hacks, rug pulls, and market crashes happen regularly.

Diversify across protocols. Even the best protocols have smart contract risk. Spreading funds limits any single protocol's impact.

Use established protocols. New protocols may offer higher yields but carry higher risk. Time-tested code is safer.

Monitor positions regularly. Collateral ratios change as prices move. Liquidation can happen quickly in volatile markets.

Keep records for taxes. Most jurisdictions tax crypto gains. DeFi's complexity makes tracking essential.

Understanding What You Use

Before depositing funds anywhere, understand the protocol. Read documentation. Review audits. Understand token economics. Know who controls upgrade keys.

If you cannot explain where yield comes from, do not chase it. Sustainable yield has identifiable sources: trading fees, borrower interest, protocol revenue. Unsustainable yield comes from token emissions that dilute value.

If a protocol requires understanding complex mechanics you do not grasp, either learn those mechanics or avoid the protocol. DeFi rewards knowledge and punishes ignorance.

SECTION 8: THE FUTURE OF DEFI

Real World Assets

DeFi is expanding beyond crypto-native assets. Real World Assets (RWAs) tokenize traditional financial instruments: treasury bills, corporate bonds, real estate, and more.

MakerDAO now holds over \$2 billion in US treasury bonds as DAI backing. Centrifuge finances real-world businesses through on-chain credit. Ondo Finance tokenizes treasury funds.

RWAs bridge traditional and decentralized finance. They bring stable yields from traditional markets while benefiting from blockchain efficiency and transparency.

Account Abstraction

Current DeFi requires users to manage private keys, pay gas in native tokens, and sign every transaction. Account abstraction enables better user experiences.

Smart contract wallets can pay gas in any token, require multiple signatures for large transactions, enable session keys for gaming, and implement social recovery. ERC-4337 standardizes these capabilities.

Better user experience expands DeFi's addressable market. Most people will not manage seed phrases. Account abstraction can make DeFi feel like traditional apps while maintaining self-custody benefits.

Cross-Chain Future

DeFi currently fragments across chains. Liquidity on Ethereum cannot directly interact with Solana. Bridges exist but add risk, cost, and complexity.

Intent-based protocols abstract away bridging. Users express what they want (swap A for B) and solvers compete to fulfill the intent, handling cross-chain mechanics invisibly.

Chain abstraction aims to make the underlying chain invisible to users. Deposit on any chain, use any protocol, withdraw anywhere. Multiple projects race toward this vision.

Institutional Adoption

Traditional finance institutions are exploring DeFi. JP Morgan tested private pool lending. BlackRock tokenized a money market fund. Major banks experiment with blockchain settlement.

Institutional adoption brings capital, credibility, and regulatory engagement. It also brings compliance requirements that may conflict with DeFi's permissionless nature.

The future likely involves multiple tracks: fully permissionless protocols serving global users and compliant versions serving regulated institutions. Both can coexist.

CHAPTER SUMMARY

DeFi reimagines financial services using smart contracts instead of institutions. It provides permissionless access, transparency, composability, and efficiency. But it also carries significant risks from smart contracts, oracles, liquidations, and regulations.

TVL measures capital deployed. Revenue measures usage. Active users measure adoption. Combined metrics reveal protocol health.

DeFi differs from traditional finance in speed, access, custody, transparency, and regulatory treatment. Each model has advantages depending on user needs.

The DeFi ecosystem spans DEXs, lending protocols, liquid staking, derivatives, and yield aggregators across multiple blockchains. Understanding this landscape helps navigate opportunities.

Getting started requires proper tools, careful first steps, rigorous risk management, and deep understanding of protocols before committing funds.

The future includes real-world assets, account abstraction, cross-chain unification, and institutional adoption. DeFi continues evolving rapidly.

With fundamentals established, the next chapter explores DEXs in depth. Automated market makers, liquidity pools, and swap routing form the core trading infrastructure of DeFi.

CHAPTER 2: DECENTRALIZED EXCHANGES

Traditional exchanges match buyers and sellers through order books. Market makers quote bid and ask prices. Trades execute when orders cross. This model requires significant infrastructure, market makers with capital, and centralized order matching.

Decentralized exchanges reimaged trading using smart contracts and algorithms. Automated Market Makers eliminated the need for order books. Liquidity pools replaced market makers. Anyone can trade instantly against protocol-

held reserves. This chapter explores how DEXs work and how to integrate them into your applications.

SECTION 1: AUTOMATED MARKET MAKER MECHANICS

The Constant Product Formula

Uniswap popularized the constant product AMM. The core formula is elegantly simple.

$$x * y = k$$

Where x is the quantity of token A, y is the quantity of token B, and k is a constant. After any trade, the product must remain unchanged.

A pool starts with 100 ETH and 300,000 USDC. The constant k equals 30,000,000. If a trader wants to buy ETH with USDC, they add USDC to the pool and remove ETH. The new quantities must still multiply to 30,000,000.

Adding 3,000 USDC makes y equal 303,000. Solving for x : $30,000,000 / 303,000 = 99.01$ ETH. The trader receives 0.99 ETH for their 3,000 USDC, a price of about \$3,030 per ETH.

The following Python code demonstrates the constant product formula.

```
class ConstantProductAMM:
    def __init__(self, reserve_a, reserve_b):
        self.reserve_a = reserve_a
        self.reserve_b = reserve_b
        self.k = reserve_a * reserve_b

    def get_price(self):
        return self.reserve_b / self.reserve_a
```



```

def get_amount_out(self, amount_in, token_in_is_a):
    if token_in_is_a:
        new_reserve_a = self.reserve_a + amount_in
        new_reserve_b = self.k / new_reserve_a
        amount_out = self.reserve_b - new_reserve_b
    else:
        new_reserve_b = self.reserve_b + amount_in
        new_reserve_a = self.k / new_reserve_b
        amount_out = self.reserve_a - new_reserve_a

    return amount_out

def swap(self, amount_in, token_in_is_a):
    amount_out = self.get_amount_out(amount_in, token_in_is_a)

    if token_in_is_a:
        self.reserve_a += amount_in
        self.reserve_b -= amount_out
    else:
        self.reserve_b += amount_in
        self.reserve_a -= amount_out

    return amount_out

```

Price Impact and Slippage

Large trades relative to pool size experience significant price impact. The constant product formula means each unit purchased costs more than the last.

Buying 1 ETH from a pool with 100 ETH and 300,000 USDC costs about 3,030 USDC. Buying 10 ETH costs about 33,300 USDC total, averaging 3,330 per ETH. Buying 50 ETH costs about 300,000 USDC, or 6,000 per ETH.

Slippage is the difference between expected and executed

price. Traders set slippage tolerance to limit losses from price movement during transaction confirmation.

The following code calculates price impact.

```
def calculate_price_impact(pool, amount_in, token_in_is_a):
    spot_price = pool.get_price() if token_in_is_a else 1 /
    pool.get_price()

    amount_out = pool.get_amount_out(amount_in, token_in_is_a)

    execution_price = amount_in / amount_out if token_in_is_a
    else amount_out / amount_in

    price_impact = abs(execution_price - spot_price) / spot_price
    * 100

    return {
        "spot_price": spot_price,
        "execution_price": execution_price,
        "amount_out": amount_out,
        "price_impact_percent": price_impact
    }
```

Trading Fees

DEXs charge fees on every swap. Uniswap V2 charges 0.3% on all trades. This fee is added to pool reserves, increasing k over time and rewarding liquidity providers.

The fee adjusts the constant product formula. Instead of using the full input amount, the pool calculates output based on 99.7% of the input.

```
class AMMWithFee:
    def __init__(self, reserve_a, reserve_b, fee_percent = 0.3):
```

```

self.reserve_a = reserve_a
self.reserve_b = reserve_b
self.fee_percent = fee_percent

def get_amount_out(self, amount_in, token_in_is_a):
    fee = amount_in * self.fee_percent / 100
    amount_in_after_fee = amount_in - fee

    if token_in_is_a:
        numerator = amount_in_after_fee * self.reserve_b
        denominator = self.reserve_a + amount_in_after_fee
        amount_out = numerator / denominator
    else:
        numerator = amount_in_after_fee * self.reserve_a
        denominator = self.reserve_b + amount_in_after_fee
        amount_out = numerator / denominator

    return amount_out

```

SECTION 2: LIQUIDITY POOLS

Providing Liquidity

Liquidity providers deposit tokens into pools. In return, they receive LP tokens representing their share of the pool. As trades occur and fees accumulate, the value of LP tokens increases.

Initial liquidity sets the pool's price ratio. If you deposit 10 ETH and 30,000 USDC, the initial price is 3,000 USDC per ETH. Subsequent deposits must match the current ratio to avoid giving arbitrageurs free money.

The following Solidity code shows a simplified liquidity addition.

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";  
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

```
contract SimpleLiquidityPool is ERC20 {  
    IERC20 public tokenA;  
    IERC20 public tokenB;  
    uint256 public reserveA;  
    uint256 public reserveB;
```

```
    constructor(address _tokenA, address _tokenB) ERC20("LP  
Token", "LP") {  
        tokenA = IERC20(_tokenA);  
        tokenB = IERC20(_tokenB);  
    }
```

```
    function addLiquidity(uint256 amountA, uint256 amountB)  
    external returns (uint256 liquidity) {  
        tokenA.transferFrom(msg.sender, address(this), amountA);  
        tokenB.transferFrom(msg.sender, address(this), amountB);
```

```
        if (totalSupply() == 0) {  
            liquidity = sqrt(amountA * amountB);  
        } else {  
            uint256 liquidityA = (amountA * totalSupply()) / reserveA;  
            uint256 liquidityB = (amountB * totalSupply()) / reserveB;  
            liquidity = liquidityA < liquidityB ? liquidityA : liquidityB;  
        }
```

```
        _mint(msg.sender, liquidity);
```

```
        reserveA += amountA;  
        reserveB += amountB;  
    }
```

```
    function removeLiquidity(uint256 liquidity) external returns
```

```
(uint256 amountA, uint256 amountB) {
    amountA = (liquidity * reserveA) / totalSupply();
    amountB = (liquidity * reserveB) / totalSupply();
```

```
_burn(msg.sender, liquidity);
```

```
reserveA -= amountA;
```

```
reserveB -= amountB;
```

```
tokenA.transfer(msg.sender, amountA);
```

```
tokenB.transfer(msg.sender, amountB);
```

```
}
```

```
function sqrt(uint256 x) internal pure returns (uint256 y) {
```

```
    uint256 z = (x + 1) / 2;
```

```
    y = x;
```

```
    while (z < y) {
```

```
        y = z;
```

```
        z = (x / z + z) / 2;
```

```
    }
```

```
}
```

```
}
```

LP Token Value

LP tokens represent proportional ownership of pool reserves. If you hold 10% of LP tokens, you can redeem them for 10% of each token in the pool.

The value of LP tokens changes based on trading fees and price movements. Fees add value by increasing reserves without minting new LP tokens. Price movements can reduce value through impermanent loss.

```
def calculate_lp_value(reserve_a, reserve_b, total_supply,
    lp_balance, price_a, price_b):
```

$\text{share} = \text{lp_balance} / \text{total_supply}$

$\text{token_a_amount} = \text{reserve_a} * \text{share}$

$\text{token_b_amount} = \text{reserve_b} * \text{share}$

$\text{value} = \text{token_a_amount} * \text{price_a} + \text{token_b_amount} * \text{price_b}$

```
return {  
  "token_a": token_a_amount,  
  "token_b": token_b_amount,  
  "total_value": value,  
  "share_percent": share * 100  
}
```

Concentrated Liquidity

Uniswap V3 introduced concentrated liquidity. Instead of providing liquidity across all possible prices (zero to infinity), LPs choose a price range. This concentrates capital where trading actually occurs.

A liquidity provider might choose to provide liquidity between \$2,500 and \$3,500 for ETH/USDC. If ETH trades within this range, their capital is 5-10x more efficient than V2's full-range liquidity. But if price moves outside their range, they earn no fees.

Concentrated liquidity requires active management. As prices move, LPs must rebalance positions to stay in range. This creates opportunities for active managers and challenges for passive investors.

Fee Tiers

Uniswap V3 offers multiple fee tiers: 0.01%, 0.05%, 0.3%, and 1%. Different pairs suit different tiers.

Stablecoin pairs like USDC/USDT use 0.01% because price barely moves and traders are fee-sensitive.

Major pairs like ETH/USDC use 0.05% or 0.3% balancing volume with LP returns.

Exotic pairs with low liquidity use 1% to compensate LPs for impermanent loss risk.

Fee tier selection affects both traders (seeking lowest fees) and LPs (seeking highest returns).

SECTION 3: UNDERSTANDING IMPERMANENT LOSS

The Mechanics

Impermanent loss occurs when the price ratio of pooled assets changes from deposit time. The AMM's constant product formula automatically rebalances the pool, selling the appreciating asset and buying the depreciating one.

If you deposit 1 ETH and 3,000 USDC when ETH is \$3,000, and later ETH rises to \$4,000, the pool rebalances. You now have less ETH and more USDC than if you had simply held both assets.

The following code calculates impermanent loss.

```
import math
```

```
def calculate_impermanent_loss(price_ratio_change):  
    sqrt_ratio = math.sqrt(price_ratio_change)  
    il = 2 * sqrt_ratio / (1 + price_ratio_change) - 1  
    return il * 100
```

```

def detailed_impermanent_loss(
    initial_amount_a,
    initial_amount_b,
    initial_price_a,
    final_price_a,
    price_b = 1
):
    k = initial_amount_a * initial_amount_b

    initial_ratio = initial_price_a / price_b
    final_ratio = final_price_a / price_b

    final_amount_a = math.sqrt(k / final_ratio)
    final_amount_b = math.sqrt(k * final_ratio)

    pool_value = final_amount_a * final_price_a + final_amount_b
    * price_b

    hold_value = initial_amount_a * final_price_a +
    initial_amount_b * price_b

    il_absolute = pool_value - hold_value
    il_percent = (il_absolute / hold_value) * 100

    return {
        "pool_token_a": final_amount_a,
        "pool_token_b": final_amount_b,
        "pool_value": pool_value,
        "hold_value": hold_value,
        "impermanent_loss_absolute": il_absolute,
        "impermanent_loss_percent": il_percent
    }

```

Impermanent Loss Table

Understanding the magnitude helps decision-making.

Price change of 1.25x (25% increase): 0.6% impermanent loss

Price change of 1.50x (50% increase): 2.0% impermanent loss

Price change of 2.00x (100% increase): 5.7% impermanent loss

Price change of 3.00x (200% increase): 13.4% impermanent loss

Price change of 4.00x (300% increase): 20.0% impermanent loss

Price change of 5.00x (400% increase): 25.5% impermanent loss

These losses are relative to simply holding. Fee income can offset losses, but volatile pairs in low-volume pools often result in net losses for LPs.

When Impermanent Loss Becomes Permanent

Impermanent loss reverses if prices return to original ratios. But if you withdraw at divergent prices, the loss crystallizes.

In practice, most price movements are permanent. Assets trend up or down, rarely returning to exact original prices. Calling it "impermanent" is somewhat misleading.

Strategies to manage impermanent loss include providing liquidity for stable pairs, choosing high-volume pools where fees offset losses, using concentrated liquidity to limit exposure ranges, and actively managing positions.

SECTION 4: UNISWAP INTEGRATION

Uniswap V3 Architecture

Uniswap V3 consists of several contracts.

Factory creates and tracks all pools. Each unique token pair and fee tier has one pool.

Pool contracts hold liquidity and execute swaps. They implement the concentrated liquidity math.

NonfungiblePositionManager manages LP positions as NFTs. Each position has unique price bounds.

SwapRouter handles swap execution with support for multi-hop routes and exact input or output amounts.

Executing Swaps

The following code demonstrates swapping on Uniswap V3 using ethers.js.

```
import { ethers } from 'ethers'

const SWAP_ROUTER_ADDRESS =
'0xE592427A0AEce92De3Edee1F18E0157C05861564'

const SWAP_ROUTER_ABI = [
  'function exactInputSingle((address tokenIn, address tokenOut, uint24 fee, address recipient, uint256 deadline, uint256 amountIn, uint256 amountOutMinimum, uint160 sqrtPriceLimitX96)) external payable returns (uint256 amountOut)',
  'function exactOutputSingle((address tokenIn, address tokenOut, uint24 fee, address recipient, uint256 deadline, uint256 amountOut, uint256 amountInMaximum, uint160 sqrtPriceLimitX96)) external payable returns (uint256 amountIn)'
]
```

```
const ERC20_ABI = [  
  'function approve(address spender, uint256 amount) returns  
(bool)',  
  'function allowance(address owner, address spender) view  
returns (uint256)',  
  'function balanceOf(address account) view returns (uint256)'  
]
```

```
async function swapExactInput(  
  wallet,  
  tokenIn,  
  tokenOut,  
  fee,  
  amountIn,  
  minAmountOut,  
  deadline  
) {  
  const router = new  
ethers.Contract(SWAP_ROUTER_ADDRESS,  
SWAP_ROUTER_ABI, wallet)  
  const tokenContract = new ethers.Contract(tokenIn,  
ERC20_ABI, wallet)  
  
  const currentAllowance = await  
tokenContract.allowance(wallet.address,  
SWAP_ROUTER_ADDRESS)  
  
  if (currentAllowance < amountIn) {  
    const approveTx = await  
tokenContract.approve(SWAP_ROUTER_ADDRESS, amountIn)  
    await approveTx.wait()  
  }  
  
  const params = {  
    tokenIn: tokenIn,  
    tokenOut: tokenOut,
```

```

fee: fee,
recipient: wallet.address,
deadline: deadline,
amountIn: amountIn,
amountOutMinimum: minAmountOut,
sqrtPriceLimitX96: 0
}

const tx = await router.exactInputSingle(params)
const receipt = await tx.wait()

return receipt
}

```

Multi-Hop Swaps

Some token pairs lack direct pools. Multi-hop routing swaps through intermediate tokens.

```

const SWAP_ROUTER_ABI_MULTIHOP = [
  'function exactInput((bytes path, address recipient, uint256
  deadline, uint256 amountIn, uint256 amountOutMinimum))
  external payable returns (uint256 amountOut)'
]

function encodePath(tokens, fees) {
  let path = tokens[0].toLowerCase().slice(2)

  for (let i = 0; i < fees.length; i++) {
    path += fees[i].toString(16).padStart(6, '0')
    path += tokens[i + 1].toLowerCase().slice(2)
  }

  return '0x' + path
}

```

```

async function swapMultiHop(
  wallet,
  tokenPath,
  feePath,
  amountIn,
  minAmountOut,
  deadline
) {
  const router = new
ethers.Contract(SWAP_ROUTER_ADDRESS,
SWAP_ROUTER_ABI_MULTIHOP, wallet)

  const path = encodePath(tokenPath, feePath)

  const params = {
    path: path,
    recipient: wallet.address,
    deadline: deadline,
    amountIn: amountIn,
    amountOutMinimum: minAmountOut
  }

  const tx = await router.exactInput(params)
  const receipt = await tx.wait()

  return receipt
}

```

Quoting Prices

Use the Quoter contract to estimate swap outputs without executing.

```

const QUOTER_ADDRESS =
'0xb27308f9F90D607463bb33eA1BeBb41C27CE5AB6'

```

```
const QUOTER_ABI = [  
  'function quoteExactInputSingle(address tokenIn, address  
tokenOut, uint24 fee, uint256 amountIn, uint160  
sqrtPriceLimitX96) external returns (uint256 amountOut)',  
  'function quoteExactInput(bytes path, uint256 amountIn)  
external returns (uint256 amountOut)'  
]
```

```
async function getQuote(provider, tokenIn, tokenOut, fee,  
amountIn) {  
  const quoter = new ethers.Contract(QUOTER_ADDRESS,  
QUOTER_ABI, provider)
```

```
  const amountOut = await  
quoter.quoteExactInputSingle.staticCall(  
tokenIn,  
tokenOut,  
fee,  
amountIn,  
0  
)
```

```
  return amountOut  
}
```

```
async function getMultiHopQuote(provider, tokenPath,  
feePath, amountIn) {  
  const quoter = new ethers.Contract(QUOTER_ADDRESS,  
QUOTER_ABI, provider)
```

```
  const path = encodePath(tokenPath, feePath)
```

```
  const amountOut = await  
quoter.quoteExactInput.staticCall(path, amountIn)
```

```
  return amountOut  
}
```

Adding Liquidity

Adding concentrated liquidity requires specifying price bounds.

```
const POSITION_MANAGER_ADDRESS =  
'0xC36442b4a4522E871399CD717aBDD847Ab11FE88'
```

```
const POSITION_MANAGER_ABI = [  
  'function mint((address token0, address token1, uint24 fee,  
  int24 tickLower, int24 tickUpper, uint256 amount0Desired,  
  uint256 amount1Desired, uint256 amount0Min, uint256  
  amount1Min, address recipient, uint256 deadline)) external  
  payable returns (uint256 tokenId, uint128 liquidity, uint256  
  amount0, uint256 amount1)'  
]
```

```
function priceToTick(price, token0Decimals, token1Decimals)  
{  
  const adjustedPrice = price * Math.pow(10, token0Decimals  
  - token1Decimals)  
  const tick = Math.floor(Math.log(adjustedPrice) /  
  Math.log(1.0001))  
  return tick  
}
```

```
function roundTickToSpacing(tick, tickSpacing) {  
  return Math.floor(tick / tickSpacing) * tickSpacing  
}
```

```
async function addLiquidity(  
  wallet,  
  token0,  
  token1,  
  fee,
```

```

amount0,
amount1,
priceLower,
priceUpper,
deadline
) {
  const positionManager = new ethers.Contract(
    POSITION_MANAGER_ADDRESS,
    POSITION_MANAGER_ABI,
    wallet
  )

  const tickSpacing = fee === 500 ? 10 : fee === 3000 ?
60 : 200

  const tickLower =
roundTickToSpacing(priceToTick(priceLower, 18, 6),
tickSpacing)
  const tickUpper =
roundTickToSpacing(priceToTick(priceUpper, 18, 6),
tickSpacing)

  const params = {
    token0: token0,
    token1: token1,
    fee: fee,
    tickLower: tickLower,
    tickUpper: tickUpper,
    amount0Desired: amount0,
    amount1Desired: amount1,
    amount0Min: 0,
    amount1Min: 0,
    recipient: wallet.address,
    deadline: deadline
  }

  const tx = await positionManager.mint(params)

```



```
const receipt = await tx.wait()

return receipt
}
```

SECTION 5: SWAP ROUTING

Why Routing Matters

A direct swap might not give the best price. Routing through intermediate pools, splitting across multiple paths, or using different protocols can improve execution.

Consider swapping LINK to USDT. A direct LINK/USDT pool might have thin liquidity. Routing LINK to ETH to USDT through deeper pools could yield better prices despite multiple hops.

Building a Simple Router

The following code finds the best route among options.

```
class SimpleRouter {
  constructor(pools) {
    this.pools = pools
    this.graph = this.buildGraph()
  }

  buildGraph() {
    const graph = {}

    for (const pool of this.pools) {
      if (!graph[pool.tokenA]) graph[pool.tokenA] = []
      if (!graph[pool.tokenB]) graph[pool.tokenB] = []

      graph[pool.tokenA].push({
```

```
token: pool.tokenB,  
pool: pool  
})  
graph[pool.tokenB].push({  
token: pool.tokenA,  
pool: pool  
})  
}
```

```
return graph  
}
```

```
findRoutes(tokenIn, tokenOut, maxHops = 3) {  
  const routes = []
```

```
  const dfs = (current, path, visited) => {  
    if (current === tokenOut) {  
      routes.push([...path])  
      return  
    }
```

```
    if (path.length >= maxHops) return
```

```
    const neighbors = this.graph[current] || []
```

```
    for (const neighbor of neighbors) {  
      const poolKey = `${neighbor.pool.tokenA}-  
${neighbor.pool.tokenB}`
```

```
      if (visited.has(poolKey)) continue
```

```
      visited.add(poolKey)  
      path.push({ token: neighbor.token, pool: neighbor.pool })
```

```
      dfs(neighbor.token, path, visited)
```

```
      path.pop()
```

```
visited.delete(poolKey)
}  
}
```

```
dfs(tokenIn, [], new Set())
```

```
return routes  
}
```

```
calculateRouteOutput(route, amountIn, tokenIn) {  
  let currentAmount = amountIn  
  let currentToken = tokenIn
```

```
  for (const step of route) {  
    const tokenInIsA = step.pool.tokenA === currentToken
```

```
    currentAmount = step.pool.get_amount_out(currentAmount,  
    tokenInIsA)  
    currentToken = step.token  
  }
```

```
  return currentAmount  
}
```

```
findBestRoute(tokenIn, tokenOut, amountIn) {  
  const routes = this.findRoutes(tokenIn, tokenOut)
```

```
  let bestRoute = null  
  let bestOutput = 0
```

```
  for (const route of routes) {  
    const output = this.calculateRouteOutput(route, amountIn,  
    tokenIn)
```

```
    if (output > bestOutput) {  
      bestOutput = output  
      bestRoute = route
```

```

    }
  }

  return {
    route: bestRoute,
    amountOut: bestOutput
  }
}
}

```

Split Routing

Large trades benefit from splitting across multiple routes. This reduces price impact on any single pool.

```

class SplitRouter {
  constructor(router) {
    this.router = router
  }

  findOptimalSplit(tokenIn, tokenOut, amountIn, splitCount =
5) {
    const routes = this.router.findRoutes(tokenIn, tokenOut)

    if (routes.length < 2) {
      return [{
        route: routes[0],
        amount: amountIn,
        output: this.router.calculateRouteOutput(routes[0],
amountIn, tokenIn)
      }]
    }

    let bestSplit = null
    let bestTotalOutput = 0

```

```

const step = amountIn / splitCount

for (let split1 = 0; split1 <= splitCount; split1++) {
  const amount1 = split1 * step
  const amount2 = amountIn - amount1

  const output1 = amount1 > 0
  ? this.router.calculateRouteOutput(routes[0], amount1,
tokenIn)
  : 0
  const output2 = amount2 > 0
  ? this.router.calculateRouteOutput(routes[1], amount2,
tokenIn)
  : 0

  const totalOutput = output1 + output2

  if (totalOutput > bestTotalOutput) {
    bestTotalOutput = totalOutput
    bestSplit = [
      { route: routes[0], amount: amount1, output: output1 },
      { route: routes[1], amount: amount2, output: output2 }
    ]
  }
}

return bestSplit.filter(s => s.amount > 0)
}
}

```

SECTION 6: DEX AGGREGATORS

What Aggregators Do

DEX aggregators query multiple exchanges and routing options to find the best execution. Instead of manually

comparing Uniswap, Sushiswap, Curve, and others, aggregators do this automatically.

Popular aggregators include 1inch, ParaSwap, 0x, and CoW Swap. Jupiter dominates Solana aggregation. Each has different routing algorithms, liquidity sources, and fee structures.

1inch Integration

1inch provides an API for quote generation and swap execution.

```
async function get1inchQuote(chainId, fromToken, toToken,
amount) {
  const url = `https://api.1inch.dev/swap/v5.2/${chainId}/
quote`
```

```
  const params = new URLSearchParams({
    src: fromToken,
    dst: toToken,
    amount: amount
  })
```

```
  const response = await fetch(`${url}?${params}`, {
    headers: {
      'Authorization': `Bearer ${process.env.ONEINCH_API_KEY}`
    }
  })
```

```
  const data = await response.json()
```

```
  return {
    fromToken: data.fromToken,
    toToken: data.toToken,
    fromAmount: data.fromTokenAmount,
```

```
toAmount: data.toTokenAmount,  
protocols: data.protocols  
}  
}
```

```
async function get1inchSwapData(chainId, fromToken,  
toToken, amount, fromAddress, slippage) {  
  const url = `https://api.1inch.dev/swap/v5.2/${chainId}/  
swap`
```

```
  const params = new URLSearchParams({  
    src: fromToken,  
    dst: toToken,  
    amount: amount,  
    from: fromAddress,  
    slippage: slippage  
  })
```

```
  const response = await fetch(`${url}?${params}`, {  
    headers: {  
      'Authorization': `Bearer ${process.env.ONEINCH_API_KEY}`  
    }  
  })
```

```
  const data = await response.json()
```

```
  return {  
    to: data.tx.to,  
    data: data.tx.data,  
    value: data.tx.value,  
    gasLimit: data.tx.gas  
  }  
}
```

```
async function executeAggregatorSwap(wallet, swapData) {  
  const tx = await wallet.sendTransaction({  
    to: swapData.to,
```

```
data: swapData.data,  
value: swapData.value,  
gasLimit: swapData.gasLimit  
})
```

```
const receipt = await tx.wait()  
return receipt  
}
```

0x Protocol Integration

0x offers a swap API with competitive pricing.

```
async function get0xQuote(chainId, sellToken, buyToken,  
sellAmount) {  
  const baseUrl = {  
    1: 'https://api.0x.org',  
    137: 'https://polygon.api.0x.org',  
    42161: 'https://arbitrum.api.0x.org'  
  }[chainId]
```

```
  const url = `${baseUrl}/swap/v1/quote`
```

```
  const params = new URLSearchParams({  
    sellToken: sellToken,  
    buyToken: buyToken,  
    sellAmount: sellAmount  
  })
```

```
  const response = await fetch(`${url}?${params}`, {  
    headers: {  
      '0x-api-key': process.env.ZEROX_API_KEY  
    }  
  })
```

```
  const data = await response.json()
```



```

return {
  buyAmount: data.buyAmount,
  sellAmount: data.sellAmount,
  price: data.price,
  to: data.to,
  data: data.data,
  value: data.value,
  gas: data.gas,
  sources: data.sources
}
}

```

Comparing Aggregator Quotes

Query multiple aggregators and compare.

```

async function findBestAggregatorPrice(
  chainId,
  fromToken,
  toToken,
  amount,
  fromAddress
) {
  const quotes = await Promise.allSettled([
    get1inchQuote(chainId, fromToken, toToken, amount),
    get0xQuote(chainId, fromToken, toToken, amount)
  ])

  const validQuotes = quotes
    .filter(q => q.status === 'fulfilled')
    .map((q, i) => ({
      source: ['1inch', '0x'][i],
      quote: q.value
    }))
}

```

```

    validQuotes.sort((a, b) => {
      const amountA = BigInt(a.quote.toAmount ||
a.quote.buyAmount)
      const amountB = BigInt(b.quote.toAmount ||
b.quote.buyAmount)
      return amountB > amountA ? 1 : -1
    })

    return validQuotes[0]
  }

```

SECTION 7: BUILDING A DEX INTERFACE

Complete Swap Component

The following React component implements a complete swap interface.

```

import { useState, useEffect } from 'react'
import { ethers } from 'ethers'

const COMMON_TOKENS = {
  ETH: { address:
'0xEeeeeEeeeEeEeeEeEeEeeEEEEeeeeEeeeeeeeEEeE', decimals:
18, symbol: 'ETH' },
  WETH: { address:
'0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2',
decimals: 18, symbol: 'WETH' },
  USDC: { address:
'0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48',
decimals: 6, symbol: 'USDC' },
  USDT: { address:
'0xdAC17F958D2ee523a2206206994597C13D831ec7',
decimals: 6, symbol: 'USDT' },
  DAI: { address:
'0x6B175474E89094C44Da98b954E8cdCB5C69F33',

```

```
decimals: 18, symbol: 'DAI' }  
}
```

```
function SwapInterface({ provider, signer }) {  
  const [fromToken, setFromToken] =  
  useState(COMMON_TOKENS.ETH)  
  const [toToken, setToToken] =  
  useState(COMMON_TOKENS.USDC)  
  const [fromAmount, setFromAmount] = useState("")  
  const [toAmount, setToAmount] = useState("")  
  const [loading, setLoading] = useState(false)  
  const [quote, setQuote] = useState(null)  
  const [slippage, setSlippage] = useState(0.5)  
  
  useEffect(() => {  
    const fetchQuote = async () => {  
      if (!fromAmount || parseFloat(fromAmount) <= 0) {  
        setToAmount("")  
        setQuote(null)  
        return  
      }  
  
      setLoading(true)  
  
      try {  
        const amountIn = ethers.parseUnits(fromAmount,  
        fromToken.decimals)  
  
        const quoteData = await get1inchQuote(  
          1,  
          fromToken.address,  
          toToken.address,  
          amountIn.toString()  
        )  
  
        setQuote(quoteData)  
        setToAmount(ethers.formatUnits(quoteData.toAmount,
```

```

toToken.decimals))
} catch (error) {
  console.error('Quote error:', error)
  setToAmount("")
} finally {
  setLoading(false)
}
}

```

```

const debounce = setTimeout(fetchQuote, 500)
return () => clearTimeout(debounce)
}, [fromAmount, fromToken, toToken])

```

```

const handleSwap = async () => {
  if (!signer || !quote) return

```

```

  setLoading(true)

```

```

  try {
    const address = await signer.getAddress()
    const amountIn = ethers.parseUnits(fromAmount,
fromToken.decimals)

```

```

    const swapData = await get1inchSwapData(
      1,
      fromToken.address,
      toToken.address,
      amountIn.toString(),
      address,
      slippage
    )

```

```

    if (fromToken.address !==
'OxEeeeeEeeeEeEeeEeEeEEEEEEEEEEEEEEEEEEEE') {
      const tokenContract = new ethers.Contract(
        fromToken.address,
        ['function approve(address,uint256) returns (bool)'],

```

```
signer
)
const approveTx = await
tokenContract.approve(swapData.to, amountIn)
await approveTx.wait()
}

const tx = await signer.sendTransaction({
to: swapData.to,
data: swapData.data,
value: swapData.value
})

await tx.wait()

setFromAmount("")
setToAmount("")
setQuote(null)

} catch (error) {
console.error('Swap error:', error)
} finally {
setLoading(false)
}
}

const switchTokens = () => {
const temp = fromToken
setFromToken(toToken)
setToToken(temp)
setFromAmount(toAmount)
setToAmount(fromAmount)
}

return {
fromToken,
toToken,
```

```

fromAmount,
toAmount,
loading,
quote,
slippage,
setFromToken,
setToToken,
setFromAmount,
setSlippage,
handleSwap,
switchTokens
}
}

```

Price Impact Warning

Warn users about high price impact trades.

```

function PriceImpactWarning({ quote, fromAmount,
toAmount }) {
  if (!quote || !fromAmount || !toAmount) return null

  const spotPrice = parseFloat(quote.toAmount) /
parseFloat(quote.fromAmount)
  const executionPrice = parseFloat(toAmount) /
parseFloat(fromAmount)

  const priceImpact = Math.abs((executionPrice - spotPrice) /
spotPrice) * 100

  if (priceImpact < 1) {
    return <span style={{ color: 'green' }}>Price impact:
{priceImpact.toFixed(2)}%</span>
  }

  if (priceImpact < 3) {

```

```

return <span style={{ color: 'orange' }}> Price impact:
{priceImpact.toFixed(2)}%</span>
}

return (
  <div style={{ color: 'red', padding: '10px', border: '1px solid
red' }}>
    Warning: High price impact ({priceImpact.toFixed(2)}%)
    Consider reducing trade size or using limit orders.
  </div>
)
}

```

SECTION 8: SECURITY CONSIDERATIONS

Sandwich Attack Protection

Sandwich attacks occur when attackers front-run and back-run your transaction. They buy before you (pushing price up), let your transaction execute at worse price, then sell after you (capturing profit).

Protection strategies include setting tight slippage tolerance, using private transaction pools like Flashbots, and avoiding large visible trades in the mempool.

```

async function sendPrivateTransaction(wallet, txData) {
  const signedTx = await wallet.signTransaction(txData)

  const response = await fetch('https://relay.flashbots.net', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      jsonrpc: '2.0',
      id: 1,
      method: 'eth_sendPrivateTransaction',

```

```

params: [{
  tx: signedTx,
  maxBlockNumber: await wallet.provider.getBlockNumber()
  + 25
}]
})
})

const result = await response.json()
return result
}

```

Approval Security

Token approvals allow contracts to spend your tokens. Unlimited approvals are convenient but risky. If the approved contract is exploited, attackers can drain your tokens.

Best practices include approving only necessary amounts, revoking unused approvals, and checking approvals before large deposits.

```

async function checkApprovals(provider, owner,
tokenAddresses, spenderAddress) {
  const results = []

  for (const tokenAddress of tokenAddresses) {
    const token = new ethers.Contract(
      tokenAddress,
      ['function allowance(address,address) view returns
(uint256)'],
      provider
    )

    const allowance = await token.allowance(owner,
spenderAddress)

```



```

if (allowance > 0) {
  results.push({
    token: tokenAddress,
    allowance: allowance.toString(),
    isUnlimited: allowance === ethers.MaxUint256
  })
}
}

return results
}

async function revokeApproval(signer, tokenAddress,
spenderAddress) {
  const token = new ethers.Contract(
    tokenAddress,
    ['function approve(address,uint256) returns (bool)'],
    signer
  )

  const tx = await token.approve(spenderAddress, 0)
  await tx.wait()
}

```

Slippage Configuration

Slippage settings balance execution certainty against value extraction.

Too tight slippage causes transactions to fail when prices move slightly. Users waste gas on reverted transactions.

Too loose slippage enables front-running and sandwich attacks. Users lose value to MEV bots.

Recommended slippage varies by volatility. Stablecoins need only 0.1-0.3%. Major tokens like ETH need 0.5-1%. Volatile tokens may need 2-5%.

```
function recommendSlippage(fromToken, toToken, volatility)
{
  const isStablePair = isStablecoin(fromToken) &&
isStablecoin(toToken)

  if (isStablePair) {
    return 0.1
  }

  const isMajorPair = isMajorToken(fromToken) ||
isMajorToken(toToken)

  if (isMajorPair) {
    return volatility > 0.05 ? 1.0 : 0.5
  }

  return volatility > 0.1 ? 3.0 : 1.5
}
```

CHAPTER SUMMARY

Decentralized exchanges use automated market makers to enable permissionless trading. The constant product formula creates prices algorithmically from pool reserves.

Liquidity providers earn fees but face impermanent loss when prices diverge. Concentrated liquidity improves capital efficiency but requires active management.

Uniswap V3 is the leading DEX with concentrated liquidity, multiple fee tiers, and position management through NFTs. Integration requires understanding the SwapRouter, Quoter,

and PositionManager contracts.

Routing optimization finds the best execution path through multiple pools and protocols. Split routing reduces price impact for large trades.

DEX aggregators like 1inch and 0x query multiple sources to find best prices. Integration simplifies access to fragmented liquidity.

Security considerations include sandwich attack protection, approval management, and appropriate slippage configuration.

The next chapter explores lending protocols where users can borrow against collateral and earn interest on deposits.

CHAPTER 3: LENDING AND BORROWING PROTOCOLS

Traditional banking intermediates between depositors and borrowers. Banks accept deposits, pay interest, lend to borrowers at higher rates, and profit from the spread. DeFi lending protocols replicate this function through smart contracts, enabling anyone to lend or borrow without bank approval.

This chapter explores how lending protocols work, how to integrate with major platforms like Aave and Compound, and the mechanics of collateralization, interest rates, liquidations, and flash loans.

SECTION 1: HOW LENDING PROTOCOLS WORK

The Basic Model

Lending protocols pool deposits from lenders and make them available to borrowers. Interest rates adjust algorithmically

based on supply and demand. When utilization is high (many borrowers, few lenders), rates rise to attract deposits. When utilization is low, rates fall to encourage borrowing.

Unlike traditional banks, DeFi lending requires overcollateralization. Borrowers must deposit more value than they borrow. If you want to borrow \$1,000 of USDC, you might need to deposit \$1,500 of ETH as collateral. This protects lenders from default since the protocol can sell collateral to recover loans.

Anonymous, permissionless borrowing requires this overcollateralization. Banks can use credit scores, income verification, and legal recourse. DeFi has only the collateral you provide.

Supply and Borrow Mechanics

When you supply assets, you receive tokens representing your deposit plus accrued interest. Compound issues cTokens. Aave issues aTokens. These tokens increase in value over time as interest accumulates.

Depositing 100 USDC into Compound gives you approximately 100 cUSDC (the exact amount depends on the exchange rate). As borrowers pay interest, the cUSDC to USDC exchange rate increases. Your 100 cUSDC might redeem for 105 USDC after a year.

Borrowing works in reverse. You deposit collateral, then borrow up to a percentage of your collateral value. This percentage is the loan-to-value (LTV) ratio. An 80% LTV means you can borrow \$800 against \$1,000 collateral.

The following code demonstrates the basic lending pool concept.

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";  
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

```
contract SimpleLendingPool is ERC20 {  
    IERC20 public underlyingToken;  
    uint256 public totalBorrows;  
    uint256 public borrowIndex;  
    uint256 public lastAccrualTime;  
    uint256 public baseRate;  
    uint256 public multiplier;
```

```
    mapping(address => uint256) public borrowBalance;  
    mapping(address => uint256) public borrowIndex_snapshot;
```

```
    constructor(address _underlying) ERC20("Pool Token",  
"pTKN") {  
        underlyingToken = IERC20(_underlying);  
        borrowIndex = 1e18;  
        lastAccrualTime = block.timestamp;  
        baseRate = 0.02e18;  
        multiplier = 0.2e18;  
    }
```

```
    function supply(uint256 amount) external {  
        accrueInterest();
```

```
        underlyingToken.transferFrom(msg.sender, address(this),  
amount);
```

```
        uint256 exchangeRate = getExchangeRate();  
        uint256 mintAmount = (amount * 1e18) / exchangeRate;
```

```
        _mint(msg.sender, mintAmount);  
    }
```

```
function withdraw(uint256 poolTokenAmount) external {  
    accrueInterest();
```

```
    uint256 exchangeRate = getExchangeRate();  
    uint256 underlyingAmount = (poolTokenAmount *  
exchangeRate) / 1e18;
```

```
    _burn(msg.sender, poolTokenAmount);  
    underlyingToken.transfer(msg.sender, underlyingAmount);  
}
```

```
function getExchangeRate() public view returns (uint256) {  
    uint256 totalSupply_ = totalSupply();  
    if (totalSupply_ == 0) {  
        return 1e18;  
    }
```

```
    uint256 totalCash =  
underlyingToken.balanceOf(address(this));  
    uint256 totalAssets = totalCash + totalBorrows;
```

```
    return (totalAssets * 1e18) / totalSupply_;  
}
```

```
function getUtilizationRate() public view returns (uint256) {  
    uint256 totalCash =  
underlyingToken.balanceOf(address(this));  
    uint256 totalAssets = totalCash + totalBorrows;
```

```
    if (totalAssets == 0) return 0;
```

```
    return (totalBorrows * 1e18) / totalAssets;  
}
```

```
function getBorrowRate() public view returns (uint256) {  
    uint256 utilization = getUtilizationRate();
```

```

return baseRate + (utilization * multiplier) / 1e18;
}

function accrueInterest() public {
    uint256 currentTime = block.timestamp;
    uint256 timeDelta = currentTime - lastAccrualTime;

    if (timeDelta == 0) return;

    uint256 borrowRate = getBorrowRate();
    uint256 interestFactor = (borrowRate * timeDelta) / 365
    days;

    uint256 interestAccumulated = (totalBorrows *
    interestFactor) / 1e18;
    totalBorrows += interestAccumulated;

    borrowIndex = borrowIndex + (borrowIndex *
    interestFactor) / 1e18;
    lastAccrualTime = currentTime;
}
}

```

Interest-Bearing Tokens

Interest-bearing tokens simplify accounting. Instead of tracking interest for each user separately, the protocol tracks a global exchange rate that all holders share.

When you deposit, you receive tokens at the current exchange rate. As interest accrues, the exchange rate increases. When you withdraw, you redeem at the new higher rate.

This design has important benefits. Interest-bearing tokens are composable. You can use cUSDC or aUSDC in other protocols, earning lending interest while simultaneously using them as

collateral or liquidity.

SECTION 2: COMPOUND PROTOCOL

Compound Architecture

Compound pioneered the lending pool model that most protocols now follow. Each supported asset has its own market contract (cToken). The Comptroller contract manages cross-market logic like collateral factors and account liquidity.

Key Compound contracts on Ethereum mainnet include.

Comptroller:

0x3d9819210A31b4961b30EF54bE2aeD79B9c9Cd3B

cETH: 0x4Ddc2D193948926D02f9B1fE9e1daa0718270ED5

cUSDC: 0x39AA39c021dfbaE8faC545936693aC917d5E7563

cDAI: 0x5d3a536E4D6DbD6114cc1Ead35777bAB948E3643

Supplying to Compound

The following code deposits assets to Compound.

```
const CTOKEN_ABI = [  
  'function mint(uint256 mintAmount) external returns  
(uint256)',  
  'function redeem(uint256 redeemTokens) external returns  
(uint256)',  
  'function redeemUnderlying(uint256 redeemAmount)  
external returns (uint256)',  
  'function balanceOf(address owner) external view returns  
(uint256)',  
  'function balanceOfUnderlying(address owner) external  
returns (uint256)',  
  'function exchangeRateCurrent() external returns (uint256)',
```



```
'function exchangeRateStored() external view returns
(uint256)'
]
```

```
const ERC20_ABI = [
  'function approve(address spender, uint256 amount) external
returns (bool)',
  'function balanceOf(address owner) external view returns
(uint256)'
]
```

```
async function supplyToCompound(signer, underlyingAddress,
cTokenAddress, amount) {
  const underlying = new ethers.Contract(underlyingAddress,
ERC20_ABI, signer)
  const cToken = new ethers.Contract(cTokenAddress,
CTOKEN_ABI, signer)
```

```
  const approveTx = await
underlying.approve(cTokenAddress, amount)
  await approveTx.wait()
```

```
  const mintTx = await cToken.mint(amount)
  const receipt = await mintTx.wait()
```

```
  const cTokenBalance = await cToken.balanceOf(await
signer.getAddress())
```

```
  return {
    transactionHash: receipt.hash,
    cTokensReceived: cTokenBalance.toString()
  }
}
```

```
async function withdrawFromCompound(signer,
cTokenAddress, cTokenAmount) {
  const cToken = new ethers.Contract(cTokenAddress,
```

CTOKEN_ABI, signer)

```
const redeemTx = await cToken.redeem(cTokenAmount)
const receipt = await redeemTx.wait()
```

```
return receipt
}
```

```
async function withdrawUnderlyingFromCompound(signer,
cTokenAddress, underlyingAmount) {
  const cToken = new ethers.Contract(cTokenAddress,
CTOKEN_ABI, signer)
```

```
  const redeemTx = await
cToken.redeemUnderlying(underlyingAmount)
  const receipt = await redeemTx.wait()
```

```
  return receipt
}
```

Borrowing from Compound

Borrowing requires first enabling assets as collateral.

```
const COMPTROLLER_ABI = [
  'function enterMarkets(address[] calldata cTokens) external
returns (uint256[])',
  'function exitMarket(address cToken) external returns
(uint256)',
  'function getAccountLiquidity(address account) external view
returns (uint256, uint256, uint256)',
  'function markets(address cToken) external view returns
(bool, uint256, bool)'
]
```

```
const CBORROW_ABI = [
```

```

'function borrow(uint256 borrowAmount) external returns
(uint256)',
'function repayBorrow(uint256 repayAmount) external
returns (uint256)',
'function borrowBalanceCurrent(address account) external
returns (uint256)'
]

```

```

const COMPTROLLER_ADDRESS =
'0x3d9819210A31b4961b30EF54bE2aeD79B9c9Cd3B'

```

```

async function enableCollateral(signer, cTokenAddresses) {
  const comptroller = new
ethers.Contract(COMPTROLLER_ADDRESS,
COMPTROLLER_ABI, signer)

```

```

  const tx = await comptroller.enterMarkets(cTokenAddresses)
  const receipt = await tx.wait()

```

```

  return receipt
}

```

```

async function getAccountLiquidity(provider, account) {
  const comptroller = new
ethers.Contract(COMPTROLLER_ADDRESS,
COMPTROLLER_ABI, provider)

```

```

  const [error, liquidity, shortfall] = await
comptroller.getAccountLiquidity(account)

```

```

  return {
    error: error.toString(),
    liquidity: ethers.formatUnits(liquidity, 18),
    shortfall: ethers.formatUnits(shortfall, 18),
    canBorrow: liquidity > 0 && shortfall === 0n
  }
}

```

```

async function borrowFromCompound(signer, cTokenAddress,
borrowAmount) {
  const cToken = new ethers.Contract(cTokenAddress,
CBORROW_ABI, signer)

  const tx = await cToken.borrow(borrowAmount)
  const receipt = await tx.wait()

  return receipt
}

```

```

async function repayBorrowCompound(signer,
underlyingAddress, cTokenAddress, repayAmount) {
  const underlying = new ethers.Contract(underlyingAddress,
ERC20_ABI, signer)
  const cToken = new ethers.Contract(cTokenAddress,
CBORROW_ABI, signer)

  const approveTx = await
underlying.approve(cTokenAddress, repayAmount)
  await approveTx.wait()

  const repayTx = await cToken.repayBorrow(repayAmount)
  const receipt = await repayTx.wait()

  return receipt
}

```

SECTION 3: AAVE PROTOCOL

Aave Architecture

Aave V3 is the current version with improved capital efficiency and cross-chain features. Key contracts include the Pool for deposits and withdrawals, and the PoolAddressesProvider for discovering contract addresses.

Aave V3 addresses vary by chain. The PoolAddressesProvider gives access to current addresses.

Ethereum:

0x2f39d218133AFaB8F2B819B1066c7E434Ad94E9e

Polygon:

0xa97684ead0e402dC232d5A977953DF7ECBaB3CDb

Arbitrum:

0xa97684ead0e402dC232d5A977953DF7ECBaB3CDb

Supplying to Aave

Aave V3 uses a unified Pool contract for all operations.

```
const POOL_ADDRESSES_PROVIDER_ABI = [  
  'function getPool() external view returns (address)',  
  'function getPriceOracle() external view returns (address)'  
]
```

```
const AAVE_POOL_ABI = [  
  'function supply(address asset, uint256 amount, address  
onBehalfOf, uint16 referralCode) external',  
  'function withdraw(address asset, uint256 amount, address  
to) external returns (uint256)',  
  'function borrow(address asset, uint256 amount, uint256  
interestRateMode, uint16 referralCode, address onBehalfOf)  
external',  
  'function repay(address asset, uint256 amount, uint256  
interestRateMode, address onBehalfOf) external returns  
(uint256)',  
  'function getUserAccountData(address user) external view  
returns (uint256 totalCollateralBase, uint256 totalDebtBase,  
uint256 availableBorrowsBase, uint256  
currentLiquidationThreshold, uint256 ltv, uint256  
healthFactor)'
```

]

```
async function getAavePool(provider,
addressesProviderAddress) {
  const addressesProvider = new ethers.Contract(
    addressesProviderAddress,
    POOL_ADDRESSES_PROVIDER_ABI,
    provider
  )
```

```
  const poolAddress = await addressesProvider.getPool()
  return poolAddress
}
```

```
async function supplyToAave(signer, poolAddress,
assetAddress, amount) {
  const asset = new ethers.Contract(assetAddress, ERC20_ABI,
signer)
  const pool = new ethers.Contract(poolAddress,
AAVE_POOL_ABI, signer)
```

```
  const approveTx = await asset.approve(poolAddress,
amount)
  await approveTx.wait()
```

```
  const userAddress = await signer.getAddress()
```

```
  const supplyTx = await pool.supply(assetAddress, amount,
userAddress, 0)
  const receipt = await supplyTx.wait()
```

```
  return receipt
}
```

```
async function withdrawFromAave(signer, poolAddress,
assetAddress, amount) {
  const pool = new ethers.Contract(poolAddress,
```

```
AAVE_POOL_ABI, signer)
const userAddress = await signer.getAddress()

const withdrawTx = await pool.withdraw(assetAddress,
amount, userAddress)
const receipt = await withdrawTx.wait()

return receipt
}
```

Aave User Account Data

Aave provides comprehensive account health information.

```
async function getAaveAccountData(provider, poolAddress,
userAddress) {
```

```
  const pool = new ethers.Contract(poolAddress,
AAVE_POOL_ABI, provider)
```

```
  const [
totalCollateralBase,
totalDebtBase,
availableBorrowsBase,
currentLiquidationThreshold,
ltv,
healthFactor
] = await pool.getUserAccountData(userAddress)
```

```
  return {
totalCollateralUSD: ethers.formatUnits(totalCollateralBase,
8),
totalDebtUSD: ethers.formatUnits(totalDebtBase, 8),
availableBorrowsUSD:
ethers.formatUnits(availableBorrowsBase, 8),
liquidationThreshold: Number(currentLiquidationThreshold)
/ 100,
```

```

ltv: Number(ltv) / 100,
healthFactor: ethers.formatUnits(healthFactor, 18),
isHealthy: healthFactor > ethers.parseUnits('1', 18)
}
}

```

Borrowing from Aave

Aave supports both stable and variable interest rates.

```

async function borrowFromAave(
  signer,
  poolAddress,
  assetAddress,
  amount,
  useStableRate = false
) {
  const pool = new ethers.Contract(poolAddress,
AAVE_POOL_ABI, signer)
  const userAddress = await signer.getAddress()

  const interestRateMode = useStableRate ? 1 : 2

  const borrowTx = await pool.borrow(
    assetAddress,
    amount,
    interestRateMode,
    0,
    userAddress
  )

  const receipt = await borrowTx.wait()
  return receipt
}

async function repayAave(

```



```

signer,
poolAddress,
assetAddress,
amount,
useStableRate = false
) {
  const asset = new ethers.Contract(assetAddress, ERC20_ABI,
signer)
  const pool = new ethers.Contract(poolAddress,
AAVE_POOL_ABI, signer)
  const userAddress = await signer.getAddress()

  const approveTx = await asset.approve(poolAddress,
amount)
  await approveTx.wait()

  const interestRateMode = useStableRate ? 1 : 2

  const repayTx = await pool.repay(
assetAddress,
amount,
interestRateMode,
userAddress
  )

  const receipt = await repayTx.wait()
  return receipt
}

```

SECTION 4: COLLATERALIZATION AND HEALTH FACTORS

Loan-to-Value Ratios

Each asset has a maximum LTV determining how much you can borrow against it. Volatile assets have lower LTVs than stable assets.

ETH might have 80% LTV: \$1,000 of ETH enables borrowing up to \$800.

USDC might have 85% LTV: \$1,000 of USDC enables borrowing up to \$850.

Volatile tokens might have 50% LTV or be disabled as collateral entirely.

Different protocols set different LTVs based on their risk assessment.

Health Factor

Health factor measures how close a position is to liquidation. It is calculated as weighted collateral value divided by debt.

Health factor = (Collateral * Liquidation Threshold) / Debt

A health factor above 1 is safe. At 1, liquidation can begin. Below 1, the position is undercollateralized and will definitely be liquidated.

The following code calculates health factor.

```
def calculate_health_factor(collaterals, debts):
```

```
    weighted_collateral = 0
```

```
    for collateral in collaterals:
```

```
        value = collateral['amount'] * collateral['price']
```

```
        weighted = value * collateral['liquidation_threshold']
```

```
        weighted_collateral += weighted
```

```
    total_debt = 0
```

```
    for debt in debts:
```

```
        value = debt['amount'] * debt['price']
```

```
    total_debt += value
```

```
if total_debt == 0:  
    return float('inf')
```

```
health_factor = weighted_collateral / total_debt
```

```
return health_factor
```

```
def get_safe_borrow_amount(collaterals, existing_debt,  
    target_health_factor=1.5):  
    weighted_collateral = 0
```

```
    for collateral in collaterals:  
        value = collateral['amount'] * collateral['price']  
        weighted = value * collateral['liquidation_threshold']  
        weighted_collateral += weighted
```

```
    max_debt_for_target_hf = weighted_collateral /  
    target_health_factor
```

```
    safe_additional_borrow = max_debt_for_target_hf -  
    existing_debt
```

```
    return max(0, safe_additional_borrow)
```

Monitoring Position Health

Active borrowers must monitor positions as prices change.

```
class PositionMonitor {  
    constructor(config) {  
        this.provider = config.provider  
        this.poolAddress = config.poolAddress  
        this.userAddress = config.userAddress  
        this.warningThreshold = config.warningThreshold || 1.5  
        this.criticalThreshold = config.criticalThreshold || 1.2  
        this.onWarning = config.onWarning || console.log
```

```
this.onCritical = config.onCritical || console.log  
}
```

```
async checkHealth() {  
  const accountData = await getAaveAccountData(  
    this.provider,  
    this.poolAddress,  
    this.userAddress  
  )
```

```
  const healthFactor = parseFloat(accountData.healthFactor)
```

```
  if (healthFactor < this.criticalThreshold) {  
    this.onCritical({  
      healthFactor,  
      message: 'CRITICAL: Immediate action required',  
      accountData  
    })  
  } else if (healthFactor < this.warningThreshold) {  
    this.onWarning({  
      healthFactor,  
      message: 'WARNING: Position approaching liquidation',  
      accountData  
    })  
  }  
}
```

```
  return {  
    healthFactor,  
    status: healthFactor < this.criticalThreshold ? 'critical':  
    healthFactor < this.warningThreshold ? 'warning' : 'healthy',  
    accountData  
  }  
}
```

```
startMonitoring(intervalMs = 60000) {  
  this.intervalId = setInterval(() => this.checkHealth(),  
    intervalMs)
```

```
this.checkHealth()
}

stopMonitoring() {
  if (this.intervalId) {
    clearInterval(this.intervalId)
  }
}
}
```

SECTION 5: LIQUIDATIONS

How Liquidations Work

When a position's health factor falls below 1, liquidators can repay part of the debt and receive collateral at a discount. This discount incentivizes liquidators to maintain protocol solvency.

Aave allows liquidating up to 50% of a position's debt in one transaction. The liquidator receives collateral worth 105-110% of the debt they repay, depending on the asset.

For example, if a user has \$1,000 debt and \$950 collateral (health factor below 1), a liquidator might repay \$500 of debt and receive \$525 worth of collateral.

Liquidation Bot Architecture

Liquidation bots monitor positions and execute profitable liquidations.

```
class LiquidationBot {
  constructor(config) {
    this.provider = config.provider
    this.signer = config.signer
```

```
this.poolAddress = config.poolAddress
this.minProfit = config.minProfit || ethers.parseEther('0.01')
this.positions = new Map()
}
```

```
async scanForLiquidatable() {
  const liquidatable = []
```

```
  for (const [userAddress, position] of this.positions) {
    const accountData = await getAaveAccountData(
      this.provider,
      this.poolAddress,
      userAddress
    )
```

```
    const healthFactor = parseFloat(accountData.healthFactor)
```

```
    if (healthFactor < 1) {
      liquidatable.push({
        userAddress,
        healthFactor,
        totalDebtUSD: parseFloat(accountData.totalDebtUSD),
        totalCollateralUSD:
          parseFloat(accountData.totalCollateralUSD)
      })
    }
  }
}
```

```
  return liquidatable.sort((a, b) => a.healthFactor -
    b.healthFactor)
}
```

```
  calculateLiquidationProfit(position, debtToRepay,
    liquidationBonus) {
    const collateralReceived = debtToRepay * (1 +
      liquidationBonus)
    const profit = collateralReceived - debtToRepay
```

```

return profit
}

async executeLiquidation(
  collateralAsset,
  debtAsset,
  userAddress,
  debtToCover,
  receiveAToken = false
) {
  const pool = new ethers.Contract(this.poolAddress, [
    'function liquidationCall(address collateralAsset, address
    debtAsset, address user, uint256 debtToCover, bool
    receiveAToken) external'
  ], this.signer)

  const debtToken = new ethers.Contract(debtAsset,
    ERC20_ABI, this.signer)
  await debtToken.approve(this.poolAddress, debtToCover)

  const tx = await pool.liquidationCall(
    collateralAsset,
    debtAsset,
    userAddress,
    debtToCover,
    receiveAToken
  )

  return tx.wait()
}
}

```

Protecting Against Liquidation

Borrowers can take actions to avoid liquidation.

Repay debt directly reduces the debt side of the health factor equation.

Add collateral increases the collateral side.

Swap collateral to more stable assets reduces volatility exposure.

Use stop-loss automation to trigger protective actions at health factor thresholds.

```
async function autoProtectPosition(
  signer,
  poolAddress,
  healthFactorThreshold,
  repayAsset,
  repayAmount
) {
  const accountData = await getAaveAccountData(
    signer.provider,
    poolAddress,
    await signer.getAddress()
  )

  const healthFactor = parseFloat(accountData.healthFactor)

  if (healthFactor < healthFactorThreshold) {
    console.log(`Health factor ${healthFactor} below threshold
    ${healthFactorThreshold}`)
    console.log('Executing protective repayment...')

    const receipt = await repayAave(
      signer,
      poolAddress,
      repayAsset,
      repayAmount,
```



```

false
)

console.log('Repayment complete:', receipt.hash)

const newAccountData = await getAaveAccountData(
  signer.provider,
  poolAddress,
  await signer.getAddress()
)

console.log('New health factor:',
newAccountData.healthFactor)

return receipt
}

return null
}

```

SECTION 6: INTEREST RATE MODELS

Supply and Borrow Rates

Interest rates in lending protocols are determined algorithmically based on utilization. Higher utilization means higher rates.

$\text{Utilization} = \text{Total Borrows} / \text{Total Supply}$

$\text{Borrow Rate} = \text{Base Rate} + (\text{Utilization} * \text{Multiplier})$

$\text{Supply Rate} = \text{Borrow Rate} * \text{Utilization} * (1 - \text{Reserve Factor})$

The reserve factor captures a percentage of interest for the protocol treasury.

Kinked Interest Rate Model

Most protocols use a kinked model that sharply increases rates above an optimal utilization point. This encourages utilization near the target while preventing complete depletion.

```
class KinkedInterestRateModel:
    def __init__(
        self,
        base_rate=0.02,
        multiplier=0.1,
        jump_multiplier=0.8,
        optimal_utilization=0.8,
        reserve_factor=0.1
    ):
        self.base_rate = base_rate
        self.multiplier = multiplier
        self.jump_multiplier = jump_multiplier
        self.optimal_utilization = optimal_utilization
        self.reserve_factor = reserve_factor

    def get_borrow_rate(self, utilization):
        if utilization <= self.optimal_utilization:
            return self.base_rate + (utilization * self.multiplier)
        else:
            normal_rate = self.base_rate + (self.optimal_utilization *
            self.multiplier)
            excess_utilization = utilization - self.optimal_utilization
            return normal_rate + (excess_utilization *
            self.jump_multiplier)

    def get_supply_rate(self, utilization):
        borrow_rate = self.get_borrow_rate(utilization)
        return borrow_rate * utilization * (1 - self.reserve_factor)
```

```

def simulate_rates(self):
    results = []

    for util_percent in range(0, 101, 5):
        utilization = util_percent / 100
        borrow_rate = self.get_borrow_rate(utilization)
        supply_rate = self.get_supply_rate(utilization)

        results.append({
            'utilization': util_percent,
            'borrow_rate_apy': borrow_rate * 100,
            'supply_rate_apy': supply_rate * 100
        })

    return results

```

Stable vs Variable Rates

Aave offers both stable and variable borrow rates.

Variable rates change every block based on utilization. They are usually lower but unpredictable.

Stable rates remain fixed for the borrower until rebalancing occurs. They provide certainty but are typically higher.

The protocol can rebalance stable rates when market conditions change significantly, protecting against rate arbitrage.

Reading Current Rates

The following code fetches current rates from Aave.

```

const AAVE_DATA_PROVIDER_ABI = [
    'function getReserveData(address asset) external view returns

```

```
(uint256 unbacked, uint256 accruedToTreasuryScaled,
uint256 totalAToken, uint256 totalStableDebt, uint256
totalVariableDebt, uint256 liquidityRate, uint256
variableBorrowRate, uint256 stableBorrowRate, uint256
averageStableBorrowRate, uint256 liquidityIndex, uint256
variableBorrowIndex, uint40 lastUpdateTimestamp)'
]
```

```
async function getAaveRates(provider, dataProviderAddress,
assetAddress) {
  const dataProvider = new ethers.Contract(
    dataProviderAddress,
    AAVE_DATA_PROVIDER_ABI,
    provider
  )
```

```
  const data = await
  dataProvider.getReserveData(assetAddress)
```

```
  const rayToPercent = (ray) => {
    return (Number(ray) / 1e27) * 100
  }
```

```
  return {
    supplyAPY: rayToPercent(data.liquidityRate),
    variableBorrowAPY: rayToPercent(data.variableBorrowRate),
    stableBorrowAPY: rayToPercent(data.stableBorrowRate),
    totalSupply: data.totalAToken.toString(),
    totalVariableDebt: data.totalVariableDebt.toString(),
    totalStableDebt: data.totalStableDebt.toString()
  }
}
```

SECTION 7: FLASH LOANS

What Are Flash Loans

Flash loans allow borrowing without collateral as long as the loan is repaid within the same transaction. If repayment fails, the entire transaction reverts, meaning the loan never happened.

This atomic property enables use cases impossible in traditional finance: arbitrage across exchanges, collateral swaps, self-liquidation, and leveraged position management.

Aave Flash Loan Implementation

Aave charges a 0.05% fee on flash loans. The borrower must implement a specific interface.

```
pragma solidity ^0.8.24;
```

```
import "@aave/v3-core/contracts/flashloan/base/FlashLoanSimpleReceiverBase.sol";
import "@aave/v3-core/contracts/interfaces/IPoolAddressesProvider.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

```
contract FlashLoanExample is FlashLoanSimpleReceiverBase {
    address public owner;
```

```
    constructor(address _addressProvider)
        FlashLoanSimpleReceiverBase(IPoolAddressesProvider(_addressProvider)) {
        owner = msg.sender;
    }
```

```
    function executeFlashLoan(address asset, uint256 amount)
        external {
        require(msg.sender == owner, "Only owner");
```

```
        POOL.flashLoanSimple(
```

```

address(this),
asset,
amount,
"",
0
);
}

```

```

function executeOperation(
address asset,
uint256 amount,
uint256 premium,
address initiator,
bytes calldata params
) external override returns (bool) {
require(msg.sender == address(POOL), "Only pool");
require(initiator == address(this), "Only self-initiated");

uint256 amountOwed = amount + premium;

IERC20(asset).approve(address(POOL), amountOwed);

return true;
}
}

```

Flash Loan Arbitrage Example

The following contract demonstrates DEX arbitrage using flash loans.

```

pragma solidity ^0.8.24;

import "@aave/v3-core/contracts/flashloan/base/
FlashLoanSimpleReceiverBase.sol";
import "@aave/v3-core/contracts/interfaces/

```

```
IPoolAddressesProvider.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

```
interface IUniswapV2Router {
    function swapExactTokensForTokens(
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external returns (uint256[] memory amounts);
```

```
    function getAmountsOut(
        uint256 amountIn,
        address[] calldata path
    ) external view returns (uint256[] memory amounts);
}
```

```
contract FlashLoanArbitrage is FlashLoanSimpleReceiverBase
{
    address public owner;
    IUniswapV2Router public router1;
    IUniswapV2Router public router2;
```

```
    constructor(
        address _addressProvider,
        address _router1,
        address _router2
    )
```

```
FlashLoanSimpleReceiverBase(IPoolAddressesProvider(_addressProvider))
{
    owner = msg.sender;
    router1 = IUniswapV2Router(_router1);
    router2 = IUniswapV2Router(_router2);
}
```

```
function executeArbitrage(
```

```
address borrowAsset,  
uint256 borrowAmount,  
address intermediateAsset  
) external {  
    require(msg.sender == owner, "Only owner");
```

```
bytes memory params = abi.encode(intermediateAsset);
```

```
POOL.flashLoanSimple(  
    address(this),  
    borrowAsset,  
    borrowAmount,  
    params,  
    0  
);  
}
```

```
function executeOperation(  
    address asset,  
    uint256 amount,  
    uint256 premium,  
    address initiator,  
    bytes calldata params  
) external override returns (bool) {  
    require(msg.sender == address(POOL), "Only pool");
```

```
    address intermediateAsset = abi.decode(params, (address));
```

```
    address[] memory path1 = new address[](2);  
    path1[0] = asset;  
    path1[1] = intermediateAsset;
```

```
    IERC20(asset).approve(address(router1), amount);  
    uint256[] memory amounts1 =  
    router1.swapExactTokensForTokens(  
        amount,  
        0,
```



```
path1,  
address(this),  
block.timestamp  
);
```

```
address[] memory path2 = new address[](2);  
path2[0] = intermediateAsset;  
path2[1] = asset;
```

```
uint256 intermediateAmount = amounts1[1];  
IERC20(intermediateAsset).approve(address(router2),  
intermediateAmount);  
uint256[] memory amounts2 =  
router2.swapExactTokensForTokens(  
intermediateAmount,  
0,  
path2,  
address(this),  
block.timestamp  
);
```

```
uint256 amountOwed = amount + premium;  
uint256 finalAmount = amounts2[1];
```

```
require(finalAmount >= amountOwed, "Not profitable");
```

```
IERC20(asset).approve(address(POOL), amountOwed);
```

```
uint256 profit = finalAmount - amountOwed;  
if (profit > 0) {  
IERC20(asset).transfer(owner, profit);  
}
```

```
return true;  
}
```

```
function checkArbitrageProfit(
```

```

address assetA,
address assetB,
uint256 amount
) external view returns (int256 profit) {
address[] memory path1 = new address[](2);
path1[0] = assetA;
path1[1] = assetB;

address[] memory path2 = new address[](2);
path2[0] = assetB;
path2[1] = assetA;

uint256[] memory amounts1 =
router1.getAmountsOut(amount, path1);
uint256[] memory amounts2 =
router2.getAmountsOut(amounts1[1], path2);

uint256 flashLoanFee = (amount * 5) / 10000;
uint256 totalCost = amount + flashLoanFee;

profit = int256(amounts2[1]) - int256(totalCost);
}
}

```

Flash Loan for Collateral Swap

Flash loans enable changing collateral without closing positions.

```

pragma solidity ^0.8.24;

import "@aave/v3-core/contracts/flashloan/base/FlashLoanSimpleReceiverBase.sol";
import "@aave/v3-core/contracts/interfaces/IPoolAddressesProvider.sol";
import "@aave/v3-core/contracts/interfaces/IPool.sol";

```

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

```
interface ISwapRouter {  
    function exactInputSingle(  
        ExactInputSingleParams calldata params  
    ) external returns (uint256 amountOut);
```

```
    struct ExactInputSingleParams {  
        address tokenIn;  
        address tokenOut;  
        uint24 fee;  
        address recipient;  
        uint256 deadline;  
        uint256 amountIn;  
        uint256 amountOutMinimum;  
        uint160 sqrtPriceLimitX96;  
    }  
}
```

```
contract CollateralSwap is FlashLoanSimpleReceiverBase {  
    ISwapRouter public swapRouter;  
    address public owner;
```

```
    constructor(  
        address _addressProvider,  
        address _swapRouter  
    )
```

```
FlashLoanSimpleReceiverBase(IPoolAddressesProvider(_addressProvider)) {  
    swapRouter = ISwapRouter(_swapRouter);  
    owner = msg.sender;  
}
```

```
    function swapCollateral(  
        address currentCollateral,  
        address newCollateral,  
        uint256 borrowAmount,
```

```
uint256 withdrawAmount,  
uint24 swapFee  
) external {  
    require(msg.sender == owner, "Only owner");
```

```
    bytes memory params = abi.encode(  
        currentCollateral,  
        newCollateral,  
        withdrawAmount,  
        swapFee  
    );
```

```
    address debtAsset = currentCollateral;
```

```
    POOL.flashLoanSimple(  
        address(this),  
        debtAsset,  
        borrowAmount,  
        params,  
        0  
    );  
}
```

```
function executeOperation(  
    address asset,  
    uint256 amount,  
    uint256 premium,  
    address initiator,  
    bytes calldata params  
) external override returns (bool) {  
    (  
        address currentCollateral,  
        address newCollateral,  
        uint256 withdrawAmount,  
        uint24 swapFee  
    ) = abi.decode(params, (address, address, uint256, uint24));
```

```
IERC20(asset).approve(address(POOL), amount);  
POOL.repay(asset, amount, 2, owner);
```

```
POOL.withdraw(currentCollateral, withdrawAmount,  
address(this));
```

```
IERC20(currentCollateral).approve(address(swapRouter),  
withdrawAmount);
```

```
uint256 newCollateralAmount =  
swapRouter.exactInputSingle(  
ISwapRouter.ExactInputSingleParams({  
tokenIn: currentCollateral,  
tokenOut: newCollateral,  
fee: swapFee,  
recipient: address(this),  
deadline: block.timestamp,  
amountIn: withdrawAmount,  
amountOutMinimum: 0,  
sqrtPriceLimitX96: 0  
}))  
);
```

```
IERC20(newCollateral).approve(address(POOL),  
newCollateralAmount);  
POOL.supply(newCollateral, newCollateralAmount, owner,  
0);
```

```
uint256 amountOwed = amount + premium;  
POOL.borrow(asset, amountOwed, 2, 0, owner);
```

```
IERC20(asset).approve(address(POOL), amountOwed);
```

```
return true;  
}  
}
```

SECTION 8: BUILDING A LENDING DASHBOARD

Position Overview Component

The following code creates a comprehensive lending position view.

```
async function getLendingPositionOverview(provider,
userAddress, protocols) {
  const positions = []

  for (const protocol of protocols) {
    if (protocol.name === 'aave') {
      const aavePosition = await getAavePosition(
        provider,
        protocol.poolAddress,
        protocol.dataProvider,
        userAddress
      )
      positions.push({ protocol: 'Aave', ...aavePosition })
    }

    if (protocol.name === 'compound') {
      const compoundPosition = await getCompoundPosition(
        provider,
        protocol.comptroller,
        protocol.markets,
        userAddress
      )
      positions.push({ protocol: 'Compound', ...compoundPosition })
    }
  }

  return positions
}
```

```
async function getAavePosition(provider, poolAddress,
dataProviderAddress, userAddress) {
  const accountData = await getAaveAccountData(provider,
poolAddress, userAddress)
```

```
  const dataProvider = new ethers.Contract(
dataProviderAddress,
[
  'function getUserReserveData(address asset, address user)
external view returns (uint256 currentATokenBalance,
uint256 currentStableDebt, uint256 currentVariableDebt,
uint256 principalStableDebt, uint256 scaledVariableDebt,
uint256 stableBorrowRate, uint256 liquidityRate, uint40
stableRateLastUpdated, bool usageAsCollateralEnabled)'
],
provider
)

  return {
    healthFactor: accountData.healthFactor,
    totalCollateralUSD: accountData.totalCollateralUSD,
    totalDebtUSD: accountData.totalDebtUSD,
    availableBorrowsUSD: accountData.availableBorrowsUSD,
    ltv: accountData.ltv,
    liquidationThreshold: accountData.liquidationThreshold
  }
}
```

Interest Accrual Calculator

Calculate expected interest over time.

```
function calculateInterestAccrual(principal, rate, timeYears,
compoundingPeriods = 365) {
  const periodicRate = rate / compoundingPeriods
  const totalPeriods = compoundingPeriods * timeYears
```

```
const finalAmount = principal * Math.pow(1 +  
periodicRate, totalPeriods)  
const interest = finalAmount - principal  
  
return {  
  principal,  
  finalAmount,  
  interest,  
  apy: (Math.pow(1 + periodicRate, compoundingPeriods) - 1)  
    * 100  
}
```

```
function projectPositionValue(position, daysForward) {  
  const timeYears = daysForward / 365
```

```
  const suppliedWithInterest = calculateInterestAccrual(  
    position.suppliedUSD,  
    position.supplyAPY / 100,  
    timeYears  
  )
```

```
  const borrowedWithInterest = calculateInterestAccrual(  
    position.borrowedUSD,  
    position.borrowAPY / 100,  
    timeYears  
  )
```

```
  const netInterest = suppliedWithInterest.interest -  
    borrowedWithInterest.interest
```

```
  const projectedCollateral = position.collateralUSD  
  const projectedDebt = borrowedWithInterest.finalAmount
```

```
  const projectedHealthFactor =  
    (projectedCollateral * position.liquidationThreshold) /
```


projectedDebt

```
return {  
  daysForward,  
  suppliedWithInterest: suppliedWithInterest.finalAmount,  
  borrowedWithInterest: borrowedWithInterest.finalAmount,  
  netInterest,  
  projectedHealthFactor,  
  supplyInterestEarned: suppliedWithInterest.interest,  
  borrowInterestPaid: borrowedWithInterest.interest  
}  
}
```

Risk Analysis

Analyze position risk under price movement scenarios.

```
function analyzePositionRisk(position, priceChanges) {  
  const scenarios = []  
  
  for (const change of priceChanges) {  
    const newCollateralValue = position.collateralUSD * (1 +  
      change)  
  
    const newHealthFactor =  
      (newCollateralValue * position.liquidationThreshold) /  
      position.debtUSD  
  
    const liquidationPrice =  
      (position.debtUSD / position.liquidationThreshold) /  
      position.collateralAmount  
  
    const currentPrice = position.collateralUSD /  
      position.collateralAmount  
    const priceDrop = (currentPrice - liquidationPrice) /  
      currentPrice
```

```

scenarios.push({
  priceChange: change * 100,
  newCollateralValue,
  newHealthFactor,
  isLiquidatable: newHealthFactor < 1,
  liquidationPrice,
  safetyMargin: priceDrop * 100
})
}

return scenarios
}

```

CHAPTER SUMMARY

Lending protocols enable permissionless borrowing and lending through overcollateralized positions. Depositors earn interest from borrowers who pay rates determined by supply and demand.

Compound uses cTokens with increasing exchange rates. Aave uses aTokens and supports both stable and variable rates. Both require understanding collateral factors, health factors, and liquidation thresholds.

Collateralization protects lenders from default. Health factors measure position safety. When health factors fall below 1, liquidators can repay debt and claim discounted collateral.

Interest rate models use utilization to balance supply and demand. Kinked models sharply increase rates above optimal utilization to prevent liquidity depletion.

Flash loans enable uncollateralized borrowing within single transactions. They power arbitrage, collateral swaps, and

other atomic operations impossible in traditional finance.

Building lending integrations requires monitoring rates, tracking positions, and implementing safety measures against liquidation.

The next chapter explores yield farming and staking, building on lending mechanics to construct more complex yield-generating strategies.

CHAPTER 4: YIELD FARMING AND STAKING

Yield farming emerged as DeFi's killer application in 2020. Compound's COMP token distribution ignited a frenzy of liquidity mining programs. Users moved billions between protocols chasing the highest returns. Staking evolved alongside, offering rewards for locking tokens to secure networks or govern protocols.

This chapter explores yield generation strategies, staking contract mechanics, reward distribution systems, and the tools to evaluate whether yields are sustainable or destined to collapse.

SECTION 1: UNDERSTANDING YIELD SOURCES

Where Does Yield Come From

Every yield has a source. Understanding that source distinguishes sustainable returns from unsustainable ones.

Trading fees are sustainable yield. DEX liquidity providers earn fees from every swap. As long as trading continues, fees continue. The yield depends on volume, not token emissions.

Lending interest is sustainable yield. Borrowers pay interest to lenders. As long as borrowing demand exists, interest

payments continue.

Token emissions are often unsustainable. Many protocols distribute native tokens to attract liquidity. These rewards dilute token holders. When emissions end or token prices fall, yields collapse.

Protocol revenue sharing is sustainable if the protocol generates real revenue. Protocols that share trading fees, lending spreads, or other income with stakers provide genuine value.

The key question: who pays the yield? If the answer is "new token holders through dilution," the yield is unsustainable. If the answer is "users paying for services," the yield has fundamental backing.

Real Yield vs Inflationary Yield

Real yield comes from protocol revenue distributed to token holders. Inflationary yield comes from new token creation.

GMX pioneered the real yield narrative. The protocol shares 30% of trading fees with GMX stakers and 70% with GLP liquidity providers. These rewards come from actual trading activity, not token printing.

Calculating real yield requires knowing protocol revenue and token market cap.

```
def calculate_real_yield(annual_revenue, market_cap,
staking_ratio):
    revenue_to_stakers = annual_revenue * 0.30
    staked_value = market_cap * staking_ratio
    real_yield = (revenue_to_stakers / staked_value) * 100
    return real_yield
```

Many advertised APYs mix real yield with token emissions. A 50% APY might be 5% real yield plus 45% token emissions. When token prices drop, that 45% becomes worth less while the 5% remains stable.

The Yield Farming Cycle

Yield farming follows predictable patterns.

Launch phase brings high emissions to attract initial liquidity. APYs can exceed 1000% because few participants share large rewards.

Growth phase sees more capital enter, diluting per-dollar returns. APYs drop to 100-500% range but remain attractive.

Maturity phase reaches equilibrium. Emissions decrease according to schedule. APYs settle to 10-50% range.

Decline phase occurs when emissions end or token price collapses. Mercenary capital exits for better opportunities. TVL drops, sometimes catastrophically.

Understanding this cycle helps time entries and exits. Early participants capture highest yields but face highest risks. Late participants get lower yields but more proven protocols.

SECTION 2: LIQUIDITY MINING PROGRAMS

How Liquidity Mining Works

Protocols distribute tokens to users who provide liquidity. The more liquidity you provide, the more tokens you earn. This bootstraps initial liquidity and decentralizes token ownership.

The basic formula allocates rewards proportionally.

$$\text{Your Rewards} = (\text{Your Liquidity} / \text{Total Liquidity}) * \text{Rewards Per Period}$$

If a pool distributes 1000 tokens daily and you provide 10% of liquidity, you earn 100 tokens daily.

Implementing a Liquidity Mining Contract

The following Solidity contract implements basic liquidity mining.

```
pragma solidity ^0.8.24;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/
SafeERC20.sol";
import "@openzeppelin/contracts/security/
ReentrancyGuard.sol";

contract LiquidityMining is ReentrancyGuard {
    using SafeERC20 for IERC20;

    IERC20 public stakingToken;
    IERC20 public rewardToken;

    uint256 public rewardRate;
    uint256 public lastUpdateTime;
    uint256 public rewardPerTokenStored;
    uint256 public periodFinish;

    mapping(address => uint256) public
    userRewardPerTokenPaid;
    mapping(address => uint256) public rewards;
    mapping(address => uint256) public balances;
```

```
uint256 public totalSupply;
```

```
address public owner;
```

```
constructor(address _stakingToken, address _rewardToken) {  
    stakingToken = IERC20(_stakingToken);  
    rewardToken = IERC20(_rewardToken);  
    owner = msg.sender;  
}
```

```
modifier updateReward(address account) {  
    rewardPerTokenStored = rewardPerToken();  
    lastUpdateTime = lastTimeRewardApplicable();
```

```
    if (account != address(0)) {  
        rewards[account] = earned(account);  
        userRewardPerTokenPaid[account] =  
        rewardPerTokenStored;  
    }  
    _;  
}
```

```
function lastTimeRewardApplicable() public view returns  
(uint256) {  
    return block.timestamp < periodFinish ? block.timestamp :  
    periodFinish;  
}
```

```
function rewardPerToken() public view returns (uint256) {  
    if (totalSupply == 0) {  
        return rewardPerTokenStored;  
    }
```

```
    return rewardPerTokenStored + (  
        (lastTimeRewardApplicable() - lastUpdateTime) * rewardRate  
        * 1e18 / totalSupply  
    );
```

```
}
```

```
function earned(address account) public view returns  
(uint256) {  
    return (  
        balances[account] * (rewardPerToken() -  
        userRewardPerTokenPaid[account]) / 1e18  
    ) + rewards[account];  
}
```

```
function stake(uint256 amount) external nonReentrant  
updateReward(msg.sender) {  
    require(amount > 0, "Cannot stake 0");
```

```
    totalSupply += amount;  
    balances[msg.sender] += amount;
```

```
    stakingToken.safeTransferFrom(msg.sender, address(this),  
    amount);  
}
```

```
function withdraw(uint256 amount) external nonReentrant  
updateReward(msg.sender) {  
    require(amount > 0, "Cannot withdraw 0");  
    require(balances[msg.sender] >= amount, "Insufficient  
    balance");
```

```
    totalSupply -= amount;  
    balances[msg.sender] -= amount;
```

```
    stakingToken.safeTransfer(msg.sender, amount);  
}
```

```
function getReward() external nonReentrant  
updateReward(msg.sender) {  
    uint256 reward = rewards[msg.sender];
```



```

if (reward > 0) {
    rewards[msg.sender] = 0;
    rewardToken.safeTransfer(msg.sender, reward);
}
}

```

```

function exit() external {
    withdraw(balances[msg.sender]);
    getReward();
}

```

```

function notifyRewardAmount(uint256 reward, uint256
duration) external updateReward(address(0)) {
    require(msg.sender == owner, "Only owner");

```

```

    rewardToken.safeTransferFrom(msg.sender, address(this),
reward);

```

```

if (block.timestamp >= periodFinish) {
    rewardRate = reward / duration;
} else {
    uint256 remaining = periodFinish - block.timestamp;
    uint256 leftover = remaining * rewardRate;
    rewardRate = (reward + leftover) / duration;
}

```

```

lastUpdateTime = block.timestamp;
periodFinish = block.timestamp + duration;
}
}

```

Multiple Reward Tokens

Some programs distribute multiple reward tokens simultaneously.

```

pragma solidity ^0.8.24;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/
SafeERC20.sol";

contract MultiRewardStaking {
    using SafeERC20 for IERC20;

    IERC20 public stakingToken;

    struct RewardInfo {
        IERC20 token;
        uint256 rewardRate;
        uint256 rewardPerTokenStored;
        uint256 lastUpdateTime;
        uint256 periodFinish;
    }

    RewardInfo[] public rewardTokens;

    mapping(uint256 => mapping(address => uint256))
    public userRewardPerTokenPaid;
    mapping(uint256 => mapping(address => uint256))
    public rewards;
    mapping(address => uint256) public balances;

    uint256 public totalSupply;

    function addRewardToken(address token) external {
        rewardTokens.push(RewardInfo({
            token: IERC20(token),
            rewardRate: 0,
            rewardPerTokenStored: 0,
            lastUpdateTime: block.timestamp,
            periodFinish: block.timestamp
        }));
    }

```

```
}
```

```
function stake(uint256 amount) external {  
  _updateAllRewards(msg.sender);
```

```
  totalSupply += amount;  
  balances[msg.sender] += amount;
```

```
  stakingToken.safeTransferFrom(msg.sender, address(this),  
amount);  
}
```

```
function withdraw(uint256 amount) external {  
  _updateAllRewards(msg.sender);
```

```
  totalSupply -= amount;  
  balances[msg.sender] -= amount;
```

```
  stakingToken.safeTransfer(msg.sender, amount);  
}
```

```
function claimAllRewards() external {  
  _updateAllRewards(msg.sender);
```

```
  for (uint256 i = 0; i < rewardTokens.length; i++) {  
    uint256 reward = rewards[i][msg.sender];  
    if (reward > 0) {  
      rewards[i][msg.sender] = 0;  
      rewardTokens[i].token.safeTransfer(msg.sender, reward);  
    }  
  }  
}
```

```
function _updateAllRewards(address account) internal {  
  for (uint256 i = 0; i < rewardTokens.length; i++) {  
    RewardInfo storage info = rewardTokens[i];
```

```
info.rewardPerTokenStored = _rewardPerToken(i);  
info.lastUpdateTime = _lastTimeRewardApplicable(i);
```

```
if (account != address(0)) {  
    rewards[i][account] = _earned(i, account);  
    userRewardPerTokenPaid[i][account] =  
info.rewardPerTokenStored;  
}  
}  
}
```

```
function _rewardPerToken(uint256 index) internal view  
returns (uint256) {
```

```
    RewardInfo storage info = rewardTokens[index];
```

```
    if (totalSupply == 0) {  
        return info.rewardPerTokenStored;  
    }
```

```
    uint256 timeDelta = _lastTimeRewardApplicable(index) -  
info.lastUpdateTime;
```

```
    return info.rewardPerTokenStored + (timeDelta *  
info.rewardRate * 1e18 / totalSupply);  
}
```

```
function _lastTimeRewardApplicable(uint256 index) internal  
view returns (uint256) {  
    return block.timestamp < rewardTokens[index].periodFinish  
    ? block.timestamp  
    : rewardTokens[index].periodFinish;  
}
```

```
function _earned(uint256 index, address account) internal  
view returns (uint256) {  
    return (  
balances[account] * (_rewardPerToken(index) -
```

```

userRewardPerTokenPaid[index][account]) / 1e18
    ) + rewards[index][account];
}
}

```

SECTION 3: STAKING CONTRACTS

Simple Token Staking

Basic staking locks tokens for rewards without complex mechanics.

```

pragma solidity ^0.8.24;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/
SafeERC20.sol";

contract SimpleStaking {
    using SafeERC20 for IERC20;

    IERC20 public token;

    struct Stake {
        uint256 amount;
        uint256 timestamp;
        uint256 lockDuration;
    }

    mapping(address => Stake) public stakes;

    uint256 public totalStaked;

    uint256 public constant MIN_LOCK = 7 days;
    uint256 public constant MAX_LOCK = 365 days;

    uint256 public baseAPY = 500;

```

```
constructor(address _token) {  
    token = IERC20(_token);  
}
```

```
function stake(uint256 amount, uint256 lockDuration)  
external {  
    require(amount > 0, "Amount must be positive");  
    require(lockDuration >= MIN_LOCK && lockDuration <=  
MAX_LOCK, "Invalid lock duration");  
    require(stakes[msg.sender].amount == 0, "Already  
staking");
```

```
    token.safeTransferFrom(msg.sender, address(this), amount);
```

```
    stakes[msg.sender] = Stake({  
        amount: amount,  
        timestamp: block.timestamp,  
        lockDuration: lockDuration  
    });
```

```
    totalStaked += amount;  
}
```

```
function unstake() external {  
    Stake storage userStake = stakes[msg.sender];  
    require(userStake.amount > 0, "No stake found");  
    require(  
        block.timestamp >= userStake.timestamp +  
userStake.lockDuration,  
        "Still locked"  
    );
```

```
    uint256 reward = calculateReward(msg.sender);  
    uint256 total = userStake.amount + reward;
```

```
    totalStaked -= userStake.amount;
```

```
delete stakes[msg.sender];
```

```
token.safeTransfer(msg.sender, total);  
}
```

```
function calculateReward(address account) public view  
returns (uint256) {
```

```
    Stake storage userStake = stakes[account];  
    if (userStake.amount == 0) return 0;
```

```
    uint256 duration = block.timestamp - userStake.timestamp;  
    uint256 lockMultiplier = 100 + (userStake.lockDuration *  
100 / MAX_LOCK);
```

```
    uint256 effectiveAPY = baseAPY * lockMultiplier / 100;  
    uint256 reward = userStake.amount * effectiveAPY *  
duration / (365 days * 10000);
```

```
    return reward;  
}
```

```
function getStakeInfo(address account) external view returns  
(
```

```
    uint256 amount,  
    uint256 reward,  
    uint256 unlockTime,  
    bool canUnstake  
) {
```

```
    Stake storage userStake = stakes[account];
```

```
    amount = userStake.amount;  
    reward = calculateReward(account);  
    unlockTime = userStake.timestamp +  
userStake.lockDuration;  
    canUnstake = block.timestamp >= unlockTime;  
}  
}
```

Vote-Escrow Staking (veTokens)

Curve pioneered vote-escrow tokenomics where longer locks earn more voting power and rewards.

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";  
import "@openzeppelin/contracts/token/ERC20/utils/  
SafeERC20.sol";
```

```
contract VoteEscrow {  
    using SafeERC20 for IERC20;
```

```
    IERC20 public token;
```

```
    struct Lock {  
        uint256 amount;  
        uint256 end;  
    }
```

```
    mapping(address => Lock) public locks;
```

```
    uint256 public totalLocked;
```

```
    uint256 public constant MAXTIME = 4 * 365 days;  
    uint256 public constant WEEK = 7 days;
```

```
    constructor(address _token) {  
        token = IERC20(_token);  
    }
```

```
    function createLock(uint256 amount, uint256 unlockTime)  
    external {  
        require(amount > 0, "Amount must be positive");  
        require(locks[msg.sender].amount == 0, "Already locked");
```



```
unlockTime = (unlockTime / WEEK) * WEEK;  
require(unlockTime > block.timestamp, "Must be future");  
require(unlockTime <= block.timestamp + MAXTIME,  
"Exceeds max lock");
```

```
token.safeTransferFrom(msg.sender, address(this), amount);
```

```
locks[msg.sender] = Lock({  
  amount: amount,  
  end: unlockTime  
});
```

```
totalLocked += amount;  
}
```

```
function increaseAmount(uint256 amount) external {  
  Lock storage userLock = locks[msg.sender];  
  require(userLock.amount > 0, "No existing lock");  
  require(userLock.end > block.timestamp, "Lock expired");
```

```
token.safeTransferFrom(msg.sender, address(this), amount);
```

```
userLock.amount += amount;  
totalLocked += amount;  
}
```

```
function increaseUnlockTime(uint256 newUnlockTime)  
external {  
  Lock storage userLock = locks[msg.sender];  
  require(userLock.amount > 0, "No existing lock");  
  require(userLock.end > block.timestamp, "Lock expired");
```

```
newUnlockTime = (newUnlockTime / WEEK) * WEEK;  
require(newUnlockTime > userLock.end, "Must extend  
lock");  
require(newUnlockTime <= block.timestamp + MAXTIME,
```

```
"Exceeds max lock");
```

```
userLock.end = newUnlockTime;  
}
```

```
function withdraw() external {  
    Lock storage userLock = locks[msg.sender];  
    require(userLock.amount > 0, "No lock found");  
    require(block.timestamp >= userLock.end, "Still locked");
```

```
    uint256 amount = userLock.amount;  
    totalLocked -= amount;
```

```
    delete locks[msg.sender];
```

```
    token.safeTransfer(msg.sender, amount);  
}
```

```
function balanceOf(address account) public view returns  
(uint256) {  
    Lock storage userLock = locks[account];
```

```
    if (userLock.end <= block.timestamp) {  
        return 0;  
    }
```

```
    uint256 timeLeft = userLock.end - block.timestamp;  
    return userLock.amount * timeLeft / MAXTIME;  
}
```

```
function totalSupply() public view returns (uint256) {  
    uint256 total = 0;  
    return total;  
}  
}
```

The veToken balance decays linearly as the unlock time

approaches. A 4-year lock gives 100% voting power. A 1-year lock gives 25%. This incentivizes long-term commitment.

SECTION 4: REWARD DISTRIBUTION MECHANISMS

Merkle Drop Distribution

For large airdrops or retroactive rewards, Merkle trees enable gas-efficient claiming.

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/utils/cryptography/
MerkleProof.sol";
```

```
contract MerkleDistributor {
    IERC20 public token;
    bytes32 public merkleRoot;
```

```
    mapping(address => bool) public claimed;
```

```
    constructor(address _token, bytes32 _merkleRoot) {
        token = IERC20(_token);
        merkleRoot = _merkleRoot;
    }
```

```
    function claim(uint256 amount, bytes32[] calldata proof)
    external {
        require(!claimed[msg.sender], "Already claimed");
```

```
        bytes32 leaf = keccak256(abi.encodePacked(msg.sender,
        amount));
        require(MerkleProof.verify(proof, merkleRoot, leaf), "Invalid
        proof");
```

```
        claimed[msg.sender] = true;
```

```
token.transfer(msg.sender, amount);
}
}
```

The off-chain code generates the Merkle tree.

```
import { StandardMerkleTree } from "@openzeppelin/merkle-
tree"
import fs from 'fs'
```

```
const recipients = [
  ["0x1111111111111111111111111111111111111111111111111111111111111111",
  "1000000000000000000000000000000000000000000000000000000000000000"],
  ["0x2222222222222222222222222222222222222222222222222222222222222222",
  "5000000000000000000000000000000000000000000000000000000000000000"],
  ["0x3333333333333333333333333333333333333333333333333333333333333333",
  "2500000000000000000000000000000000000000000000000000000000000000"]
]
```

```
const tree = StandardMerkleTree.of(recipients, ["address",
"uint256"])
```

```
console.log("Merkle Root:", tree.root)
```

```
fs.writeFileSync("tree.json", JSON.stringify(tree.dump()))
```

```
function getProof(address) {
  for (const [i, v] of tree.entries()) {
    if (v[0] === address) {
      const proof = tree.getProof(i)
      return { amount: v[1], proof }
    }
  }
  return null
}
```

Epoch-Based Distribution

Some protocols distribute rewards in epochs, calculating allocations at fixed intervals.

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";  
import "@openzeppelin/contracts/token/ERC20/utils/  
SafeERC20.sol";
```

```
contract EpochRewards {  
    using SafeERC20 for IERC20;
```

```
    IERC20 public stakingToken;  
    IERC20 public rewardToken;
```

```
    uint256 public epochDuration = 7 days;  
    uint256 public currentEpoch;  
    uint256 public epochStartTime;
```

```
    mapping(uint256 => uint256) public epochRewards;  
    mapping(uint256 => uint256) public epochTotalStaked;  
    mapping(address => mapping(uint256 => uint256))  
    public userEpochStake;  
    mapping(address => uint256) public lastClaimedEpoch;  
    mapping(address => uint256) public balances;
```

```
    uint256 public totalStaked;
```

```
    constructor(address _stakingToken, address _rewardToken) {  
        stakingToken = IERC20(_stakingToken);  
        rewardToken = IERC20(_rewardToken);  
        epochStartTime = block.timestamp;  
    }
```

```
    function stake(uint256 amount) external {
```

```
_checkEpoch();
```

```
stakingToken.safeTransferFrom(msg.sender, address(this),  
amount);
```

```
balances[msg.sender] += amount;  
totalStaked += amount;
```

```
userEpochStake[msg.sender][currentEpoch] =  
balances[msg.sender];  
}
```

```
function withdraw(uint256 amount) external {  
_checkEpoch();
```

```
require(balances[msg.sender] >= amount, "Insufficient  
balance");
```

```
balances[msg.sender] -= amount;  
totalStaked -= amount;
```

```
userEpochStake[msg.sender][currentEpoch] =  
balances[msg.sender];
```

```
stakingToken.safeTransfer(msg.sender, amount);  
}
```

```
function claimRewards() external {  
_checkEpoch();
```

```
uint256 totalReward = 0;
```

```
for (uint256 epoch = lastClaimedEpoch[msg.sender]; epoch  
< currentEpoch; epoch++) {  
uint256 userStake = userEpochStake[msg.sender][epoch];  
uint256 totalEpochStake = epochTotalStaked[epoch];  
uint256 rewards = epochRewards[epoch];
```

```
if (userStake > 0 && totalEpochStake > 0) {  
    totalReward += (rewards * userStake) / totalEpochStake;  
}  
}
```

```
lastClaimedEpoch[msg.sender] = currentEpoch;
```

```
if (totalReward > 0) {  
    rewardToken.safeTransfer(msg.sender, totalReward);  
}  
}
```

```
function _checkEpoch() internal {  
    while (block.timestamp >= epochStartTime +  
epochDuration) {  
        epochTotalStaked[currentEpoch] = totalStaked;
```

```
        currentEpoch++;  
        epochStartTime += epochDuration;  
    }  
}
```

```
function addEpochRewards(uint256 epoch, uint256 amount)  
external {  
    rewardToken.safeTransferFrom(msg.sender, address(this),  
amount);  
    epochRewards[epoch] += amount;  
}  
}
```

SECTION 5: AUTO-COMPOUNDING VAULTS

How Auto-Compounding Works

Manual yield farming requires claiming rewards and reinvesting them. Auto-compounding vaults automate this

process, harvesting rewards and depositing them back to compound returns.

The difference between simple and compound interest grows dramatically over time.

```
def compare_yields(principal, apy, years,
compounds_per_year):
    simple_final = principal * (1 + apy * years)

    compound_final = principal * ((1 + apy /
compounds_per_year) ** (compounds_per_year * years))

    continuous_final = principal * (2.71828 ** (apy * years))

    return {
'simple': simple_final,
'compound': compound_final,
'continuous': continuous_final,
'compound_advantage': (compound_final / simple_final - 1) *
100
}
```

Auto-Compounding Vault Contract

The following contract implements a basic auto-compounding vault.

```
pragma solidity ^0.8.24;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/
SafeERC20.sol";

interface IFarm {
    function deposit(uint256 amount) external;
```



```

function withdraw(uint256 amount) external;
function harvest() external;
function pendingReward(address user) external view returns
(uint256);
function userInfo(address user) external view returns
(uint256 amount, uint256 rewardDebt);
}

```

```

interface IRouter {
function swapExactTokensForTokens(
uint256 amountIn,
uint256 amountOutMin,
address[] calldata path,
address to,
uint256 deadline
) external returns (uint256[] memory amounts);
}

```

```

contract AutoCompoundVault is ERC20 {
using SafeERC20 for IERC20;

```

```

IERC20 public depositToken;
IERC20 public rewardToken;
IFarm public farm;
IRouter public router;

```

```

address[] public swapPath;

```

```

uint256 public lastHarvest;
uint256 public harvestInterval = 1 hours;

```

```

uint256 public performanceFee = 300;
uint256 public constant FEE_DENOMINATOR = 10000;
address public treasury;

```

```

constructor(
address _depositToken,

```

```

address _rewardToken,
address _farm,
address _router,
address _treasury
) ERC20("Auto Compound Vault", "acVAULT") {
    depositToken = IERC20(_depositToken);
    rewardToken = IERC20(_rewardToken);
    farm = IFarm(_farm);
    router = IRouter(_router);
    treasury = _treasury;

    swapPath = new address[](2);
    swapPath[0] = _rewardToken;
    swapPath[1] = _depositToken;

    depositToken.approve(_farm, type(uint256).max);
    rewardToken.approve(_router, type(uint256).max);
}

function deposit(uint256 amount) external {
    harvest();

    uint256 totalBefore = totalDeposited();

    depositToken.safeTransferFrom(msg.sender, address(this),
    amount);

    farm.deposit(amount);

    uint256 shares;
    if (totalSupply() == 0) {
        shares = amount;
    } else {
        shares = (amount * totalSupply()) / totalBefore;
    }

    _mint(msg.sender, shares);
}

```

```
}
```

```
function withdraw(uint256 shares) external {  
    harvest();
```

```
    uint256 amount = (shares * totalDeposited()) /  
    totalSupply();
```

```
    _burn(msg.sender, shares);
```

```
    farm.withdraw(amount);
```

```
    depositToken.safeTransfer(msg.sender, amount);  
}
```

```
function harvest() public {  
    if (block.timestamp < lastHarvest + harvestInterval) {  
        return;  
    }
```

```
    uint256 pending = farm.pendingReward(address(this));  
    if (pending == 0) {  
        return;  
    }
```

```
    farm.harvest();
```

```
    uint256 rewardBalance =  
    rewardToken.balanceOf(address(this));
```

```
    uint256 fee = (rewardBalance * performanceFee) /  
    FEE_DENOMINATOR;  
    if (fee > 0) {  
        rewardToken.safeTransfer(treasury, fee);  
        rewardBalance -= fee;  
    }
```

```
    if (rewardBalance > 0) {
```

```
router.swapExactTokensForTokens(
rewardBalance,
0,
swapPath,
address(this),
block.timestamp
);
```

```
uint256 depositBalance =
depositToken.balanceOf(address(this));
if (depositBalance > 0) {
farm.deposit(depositBalance);
}
}
```

```
lastHarvest = block.timestamp;
}
```

```
function totalDeposited() public view returns (uint256) {
(uint256 deposited,) = farm.userInfo(address(this));
return deposited + depositToken.balanceOf(address(this));
}
```

```
function pricePerShare() public view returns (uint256) {
if (totalSupply() == 0) {
return 1e18;
}
return (totalDeposited() * 1e18) / totalSupply();
}
```

```
function pendingHarvest() external view returns (uint256) {
return farm.pendingReward(address(this));
}
}
```

Yearn-Style Vault Architecture

Yearn's vaults use a more sophisticated architecture with pluggable strategies.

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/
SafeERC20.sol";
```

```
interface IStrategy {
    function want() external view returns (address);
    function deposit() external;
    function withdraw(uint256 amount) external;
    function withdrawAll() external returns (uint256);
    function balanceOf() external view returns (uint256);
    function harvest() external;
}
```

```
contract YearnStyleVault is ERC20 {
    using SafeERC20 for IERC20;
```

```
    IERC20 public token;
    IStrategy public strategy;
    address public governance;
```

```
    uint256 public depositLimit;
    uint256 public managementFee = 200;
    uint256 public performanceFee = 2000;
    uint256 public constant FEE_DENOMINATOR = 10000;
```

```
    uint256 public lastReport;
```

```
    constructor(address _token) ERC20("Yearn Vault", "yVAULT")
    {
        token = IERC20(_token);
```

```
governance = msg.sender;  
depositLimit = type(uint256).max;  
}
```

```
function setStrategy(address _strategy) external {  
    require(msg.sender == governance, "Only governance");  
    require(IStrategy(_strategy).want() == address(token),  
    "Wrong token");
```

```
    if (address(strategy) != address(0)) {  
        strategy.withdrawAll();  
    }
```

```
    strategy = IStrategy(_strategy);
```

```
    uint256 balance = token.balanceOf(address(this));  
    if (balance > 0) {  
        token.safeTransfer(address(strategy), balance);  
        strategy.deposit();  
    }  
}
```

```
function deposit(uint256 amount) external returns (uint256  
shares) {  
    require(totalAssets() + amount <= depositLimit, "Deposit  
limit reached");
```

```
    uint256 totalBefore = totalAssets();
```

```
    token.safeTransferFrom(msg.sender, address(this), amount);
```

```
    if (address(strategy) != address(0)) {  
        token.safeTransfer(address(strategy), amount);  
        strategy.deposit();  
    }
```

```
    if (totalSupply() == 0) {
```

```
shares = amount;  
} else {  
shares = (amount * totalSupply()) / totalBefore;  
}
```

```
_mint(msg.sender, shares);  
}
```

```
function withdraw(uint256 shares) external returns (uint256  
amount) {  
amount = (shares * totalAssets()) / totalSupply();
```

```
_burn(msg.sender, shares);
```

```
uint256 vaultBalance = token.balanceOf(address(this));
```

```
if (amount > vaultBalance) {  
uint256 needed = amount - vaultBalance;  
strategy.withdraw(needed);  
}
```

```
token.safeTransfer(msg.sender, amount);  
}
```

```
function report(uint256 gain, uint256 loss) external {  
require(msg.sender == address(strategy), "Only strategy");
```

```
if (gain > 0) {  
uint256 fee = (gain * performanceFee) /  
FEE_DENOMINATOR;  
uint256 shares = (fee * totalSupply()) / totalAssets();  
_mint(governance, shares);  
}
```

```
lastReport = block.timestamp;  
}
```

```

function totalAssets() public view returns (uint256) {
    uint256 vaultBalance = token.balanceOf(address(this));
    uint256 strategyBalance = address(strategy) != address(0)
    ? strategy.balanceOf()
    : 0;

    return vaultBalance + strategyBalance;
}

function pricePerShare() public view returns (uint256) {
    if (totalSupply() == 0) {
        return 10 ** decimals();
    }
    return (totalAssets() * 10 ** decimals()) / totalSupply();
}
}

```

SECTION 6: RISK ASSESSMENT

Smart Contract Risk

Every yield opportunity involves smart contract risk.
Questions to evaluate.

Has the code been audited? By whom? When? Audits reduce but do not eliminate risk.

How long has the protocol been live? Time-tested protocols are safer than new launches.

What is the TVL? Higher TVL means more eyes examining the code and more at stake if bugs exist.

Is the code open source? Closed source protocols cannot be independently verified.

Are there admin keys? Who controls them? What can they do?

Timelocks reduce instant rug risk.

Impermanent Loss Risk

Liquidity provision exposes you to impermanent loss. Evaluate.

What is the historical volatility between paired assets? Higher volatility means higher IL risk.

What fee tier compensates for IL? High-fee pools can offset significant IL.

What are the additional reward emissions? Do they adequately compensate for IL?

```
def estimate_breakeven_volume(position_size, days,
il_estimate, fee_rate):
    daily_il = il_estimate / days
    daily_volume_needed = (position_size * daily_il) / fee_rate
    return daily_volume_needed
```

Economic Risk

Token emission yields are only valuable if token prices hold.

What is the emission schedule? Accelerating emissions pressure prices.

What is the token utility? Governance-only tokens have limited demand.

What is the circulating supply versus fully diluted? High FDV relative to market cap means dilution ahead.

Who holds large allocations? Team and investor unlocks can

crash prices.

```
def calculate_dilution_impact(current_price, current_supply,  
future_supply):
```

```
    dilution_ratio = future_supply / current_supply
```

```
    implied_future_price = current_price / dilution_ratio
```

```
    loss_percent = (1 - 1/dilution_ratio) * 100
```

```
    return {  
        'current_price': current_price,  
        'implied_future_price': implied_future_price,  
        'dilution_ratio': dilution_ratio,  
        'potential_loss_percent': loss_percent  
    }
```

Systemic Risk

DeFi's composability creates systemic risk. Failures cascade.

What dependencies does this protocol have? Oracle failures, bridge hacks, and stablecoin depegs can trigger losses.

What would happen if a major protocol fails? If Aave or Uniswap had issues, many dependent protocols would suffer.

Is there concentration risk? Large positions in single protocols amplify individual failures.

SECTION 7: YIELD FARMING STRATEGIES

Conservative Strategies

Low risk, modest returns. Target 5-15% APY.

Stablecoin lending on major protocols. Supply USDC or DAI to Aave or Compound. Minimal IL risk, battle-tested contracts, predictable returns.

Blue-chip LP provision. Provide liquidity to ETH/USDC or BTC/ETH pairs on Uniswap. Established pools, high volume, reasonable IL offset by fees.

Liquid staking. Hold stETH or rETH to earn Ethereum staking rewards. Protocol risk but no IL risk.

Moderate Strategies

Medium risk, enhanced returns. Target 15-40% APY.

Incentivized lending. Supply assets to newer lending protocols offering token incentives. Higher yield but less tested code.

Concentrated liquidity. Provide liquidity in narrow ranges on Uniswap V3. Higher capital efficiency and fees but requires active management.

Curve ecosystem. Provide liquidity to Curve pools, stake CRV for veCRV, boost rewards. Complex but proven system.

Aggressive Strategies

Higher risk, potentially high returns. Target 40%+ APY.

New protocol farming. Farm early in protocol launches when rewards are highest and TVL is lowest. High smart contract risk.

Leveraged yield farming. Borrow to increase position size. Amplifies gains and losses. Liquidation risk.

Cross-chain farming. Bridge assets to newer chains with higher incentives. Bridge risk plus smart contract risk.

Strategy Implementation

The following code helps evaluate and compare strategies.

```
class YieldStrategy:
    def __init__(self, name, base_apy, token_emissions_apy, risks):
        self.name = name
        self.base_apy = base_apy
        self.token_emissions_apy = token_emissions_apy
        self.risks = risks

    def total_apy(self):
        return self.base_apy + self.token_emissions_apy

    def risk_adjusted_apy(self, token_price_assumption = 0.5):
        sustainable = self.base_apy
        adjusted_emissions = self.token_emissions_apy *
            token_price_assumption

        risk_factor = 1.0
        for risk, level in self.risks.items():
            if level == 'high':
                risk_factor *= 0.7
            elif level == 'medium':
                risk_factor *= 0.85

        return (sustainable + adjusted_emissions) * risk_factor

    def compare(self, other):
        return {
            'name_comparison': f'{self.name} vs {other.name}',
            'total_apy_diff': self.total_apy() - other.total_apy(),
            'risk_adjusted_diff': self.risk_adjusted_apy() -
```

```
other.risk_adjusted_apy()  
}
```

```
def evaluate_strategies(strategies):  
    results = []  
  
    for strategy in strategies:  
        results.append({  
            'name': strategy.name,  
            'total_apy': strategy.total_apy(),  
            'risk_adjusted_apy': strategy.risk_adjusted_apy(),  
            'risks': strategy.risks  
        })  
  
    results.sort(key = lambda x: x['risk_adjusted_apy'],  
                reverse = True)  
  
    return results
```

SECTION 8: BUILDING A YIELD DASHBOARD

Aggregating Position Data

Track yields across multiple protocols.

```
async function getYieldPositions(userAddress, protocols) {  
    const positions = []  
  
    for (const protocol of protocols) {  
        try {  
            const position = await protocol.getPosition(userAddress)  
  
            if (position.balance > 0) {  
                positions.push({  
                    protocol: protocol.name,  
                    chain: protocol.chain,
```

```

token: position.token,
balance: position.balance,
valueUSD: position.valueUSD,
apy: position.apy,
pendingRewards: position.pendingRewards,
pendingRewardsUSD: position.pendingRewardsUSD
})
}
} catch (error) {
console.error(`Error fetching ${protocol.name}:`, error)
}
}

return positions
}

```

```

function calculatePortfolioMetrics(positions) {
  const totalValueUSD = positions.reduce((sum, p) => sum
+ p.valueUSD, 0)
  const totalPendingUSD = positions.reduce((sum, p) => sum
+ p.pendingRewardsUSD, 0)

  const weightedAPY = positions.reduce((sum, p) => {
    const weight = p.valueUSD / totalValueUSD
    return sum + (p.apy * weight)
  }, 0)

  const byProtocol = {}
  for (const position of positions) {
    if (!byProtocol[position.protocol]) {
      byProtocol[position.protocol] = 0
    }
    byProtocol[position.protocol] += position.valueUSD
  }

  return {
    totalValueUSD,

```

```

totalPendingUSD,
weightedAPY,
positionCount: positions.length,
protocolBreakdown: byProtocol
}
}

```

Historical Yield Tracking

Track APY changes over time.

```

class YieldTracker {
  constructor(database) {
    this.database = database
  }

  async recordYield(protocol, pool, apy, tvl, timestamp) {
    await this.database.query(
      `INSERT INTO yield_history (protocol, pool, apy, tvl,
timestamp)
VALUES ($1, $2, $3, $4, $5)`,
      [protocol, pool, apy, tvl, timestamp]
    )
  }

  async getYieldHistory(protocol, pool, days) {
    const result = await this.database.query(
      `SELECT apy, tvl, timestamp
FROM yield_history
WHERE protocol = $1 AND pool = $2
AND timestamp > NOW() - INTERVAL '${days} days'
ORDER BY timestamp ASC`,
      [protocol, pool]
    )

    return result.rows
  }
}

```

```
}
```

```
async calculateAverageAPY(protocol, pool, days) {  
  const result = await this.database.query(  
    `SELECT AVG(apy) as avg_apy, MIN(apy) as min_apy,  
    MAX(apy) as max_apy  
    FROM yield_history  
    WHERE protocol = $1 AND pool = $2  
    AND timestamp > NOW() - INTERVAL '${days} days`,  
    [protocol, pool]  
  )  
}
```

```
  return result.rows[0]  
}
```

```
async detectYieldChanges(threshold = 0.1) {  
  const result = await this.database.query(  
    `WITH recent AS (  
      SELECT protocol, pool, apy,  
      LAG(apy) OVER (PARTITION BY protocol, pool ORDER BY  
timestamp) as prev_apy  
      FROM yield_history  
      WHERE timestamp > NOW() - INTERVAL '1 day'  
    )  
    SELECT protocol, pool, apy, prev_apy,  
    (apy - prev_apy) / prev_apy as change_ratio  
    FROM recent  
    WHERE prev_apy IS NOT NULL  
    AND ABS((apy - prev_apy) / prev_apy) > $1`,  
    [threshold]  
  )  
}
```

```
  return result.rows  
}  
}
```


CHAPTER SUMMARY

Yield farming and staking provide returns through trading fees, lending interest, and token emissions. Understanding yield sources helps distinguish sustainable returns from inflationary rewards destined to decline.

Liquidity mining contracts distribute tokens proportionally to liquidity provided. The Synthetix staking model using reward-per-token tracking has become the standard implementation.

Staking contracts range from simple time-locks to sophisticated vote-escrow systems where longer locks earn greater rewards and governance power.

Auto-compounding vaults automate reward harvesting and reinvestment, dramatically improving returns over manual management through compound interest effects.

Risk assessment must evaluate smart contract security, impermanent loss exposure, token economic sustainability, and systemic dependencies.

Yield strategies span conservative stablecoin lending through aggressive new protocol farming. Risk-adjusted returns often favor simpler approaches over chasing headline APYs.

Building yield dashboards requires aggregating positions across protocols, calculating portfolio metrics, and tracking historical performance.

The next chapter explores stablecoins, the foundational assets that enable much of DeFi's lending and trading activity.

CHAPTER 5: STABLECOINS

Cryptocurrency volatility makes it unsuitable for everyday transactions. Bitcoin can swing 10% in a day. Ethereum has

moved 30% in a week. This volatility is fine for speculation but terrible for commerce, lending, and saving.

Stablecoins solve this by maintaining a stable value, typically pegged to the US dollar. They combine cryptocurrency's speed and programmability with fiat currency's stability. This chapter explores how different stablecoin designs work, their tradeoffs, integration patterns, and the risks that have caused spectacular failures.

SECTION 1: STABLECOIN CATEGORIES

The Three Pillars of Stablecoin Design

Every stablecoin must answer three questions. What backs its value? How does it maintain its peg? What happens under stress?

Different designs answer these questions differently, creating distinct categories with different risk profiles.

Fiat-collateralized stablecoins hold dollars in bank accounts. Each token is backed by one dollar in reserves. Centralized and simple.

Crypto-collateralized stablecoins lock cryptocurrency as collateral. Overcollateralization protects against volatility. Decentralized but capital inefficient.

Algorithmic stablecoins use market incentives and token mechanics to maintain peg without collateral. Capital efficient but fragile.

Hybrid designs combine elements. Real-world asset backing with on-chain transparency. Crypto collateral supplemented by algorithmic mechanisms.

Market Structure

As of late 2025, the stablecoin market exceeds \$150 billion in total supply.

USDT (Tether) dominates with approximately \$90 billion. It is fiat-collateralized and centralized.

USDC (Circle) holds second place with approximately \$25 billion. Also fiat-collateralized, but with more transparent reserves.

DAI (MakerDAO) leads decentralized stablecoins with approximately \$5 billion. Crypto-collateralized with some real-world asset backing.

FRAX started algorithmic but shifted toward collateralized design after algorithmic stablecoin failures.

Numerous smaller stablecoins serve specific chains, protocols, or use cases.

SECTION 2: FIAT-COLLATERALIZED STABLECOINS

How Fiat Backing Works

Fiat-collateralized stablecoins work simply. An issuer holds dollars in bank accounts. For every dollar deposited, they mint one stablecoin. For every stablecoin redeemed, they burn the token and release one dollar.

This creates a direct peg mechanism. If the stablecoin trades below \$1, arbitrageurs buy it cheaply and redeem for \$1, profiting from the difference and pushing price back to \$1. If it trades above \$1, arbitrageurs deposit dollars, mint stablecoins, sell them, and profit, pushing price back down.

The arbitrage mechanism only works if redemption is reliable. Trust in the issuer is essential.

USDT (Tether)

Tether launched in 2014 as the first major stablecoin. It dominates trading volume on centralized exchanges, particularly in Asia.

Tether has faced criticism for opaque reserves. Questions about whether reserves fully backed tokens led to legal settlements. Recent attestations show reserves include cash, treasury bills, commercial paper, and other investments.

USDT contract addresses differ by chain.

Ethereum:

0xdAC17F958D2ee523a2206206994597C13D831ec7

Tron: TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t

Polygon:

0xc2132D05D31c914a87C6611C10748AEb04B58e8F

Integrating USDT requires checking the specific chain's contract. Token standards vary. ERC-20 on Ethereum, TRC-20 on Tron.

USDC (Circle)

Circle launched USDC in 2018 in partnership with Coinbase. It positioned itself as the transparent, regulated alternative to Tether.

USDC publishes monthly attestations from accounting firms. Reserves are held in cash and short-term US treasuries. Circle is pursuing full banking licenses in multiple jurisdictions.

USDC contract addresses by chain.

Ethereum:

0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48

Polygon: 0x3c499c542cEF5E3811e1192ce70d8cC03d5c3359

Arbitrum:

0xaf88d065e77c8cC2239327C5EDb3A432268e5831

Base: 0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913

Circle introduced Cross-Chain Transfer Protocol (CCTP) enabling native USDC transfers across chains without bridge wrapping.

Regulatory Considerations

Fiat stablecoins face increasing regulatory scrutiny. Issuers must comply with money transmission laws, anti-money laundering requirements, and sanctions.

In 2022, Circle froze USDC associated with Tornado Cash following US Treasury sanctions. This demonstrated that fiat stablecoins are not censorship resistant. Issuers can and will freeze funds when required by law.

For applications requiring censorship resistance, fiat-backed stablecoins may not be appropriate. For mainstream applications, regulatory compliance is a feature not a bug.

SECTION 3: CRYPTO-COLLATERALIZED STABLECOINS

Overcollateralization Model

Crypto-collateralized stablecoins require depositing more value than you mint. If you deposit \$150 of ETH, you might be allowed to mint \$100 of stablecoins. This buffer protects

against collateral value decline.

The system maintains solvency through liquidations. If collateral value drops close to debt value, the system sells collateral to repay debt before it becomes undercollateralized.

This model is capital inefficient. \$150 of capital produces only \$100 of stablecoins. But it enables decentralization since no centralized issuer holds reserves.

MakerDAO and DAI

MakerDAO pioneered crypto-collateralized stablecoins with DAI in 2017. Users deposit collateral into Vaults and mint DAI against it.

Each collateral type has parameters. Liquidation ratio determines when positions can be liquidated. Stability fee is the interest rate charged on DAI debt. Debt ceiling caps how much DAI can be minted against each collateral type.

The MakerDAO system contracts on Ethereum.

DAI Token:

0x6B175474E89094C44Da98b954E8cdCB5C69F33

Vat (Core Accounting):

0x35D1b3F3D7966A1DfE207aa4514C12a259A0492B

DAI Join:

0x9759A6Ac90977b93B58547b4A71c78317f391A28

Interacting with MakerDAO requires understanding the CDP (Collateralized Debt Position) lifecycle.

```
const MAKER_ABI = {
```

```
  vatAbi: [
```

```
    'function frob(bytes32 ilk, address u, address v, address w,
```

```

int256 dink, int256 dart) external',
'function urns(bytes32 ilk, address u) external view returns
(uint256 ink, uint256 art)'
],
gemJoinAbi: [
'function join(address usr, uint256 wad) external',
'function exit(address usr, uint256 wad) external'
],
daiJoinAbi: [
'function join(address usr, uint256 wad) external',
'function exit(address usr, uint256 wad) external'
]
}

```

```

async function openVault(signer, collateralAmount,
daiAmount, ilk, gemJoin, daiJoin, vat) {
  const gemJoinContract = new ethers.Contract(gemJoin,
MAKER_ABI.gemJoinAbi, signer)
  const daiJoinContract = new ethers.Contract(daiJoin,
MAKER_ABI.daiJoinAbi, signer)
  const vatContract = new ethers.Contract(vat,
MAKER_ABI.vatAbi, signer)

```

```

  await gemJoinContract.join(await signer.getAddress(),
collateralAmount)

```

```

const dink = collateralAmount
const dart = daiAmount

```

```

  await vatContract.frob(
    ilk,
    await signer.getAddress(),
    await signer.getAddress(),
    await signer.getAddress(),
    dink,
    dart
  )

```

```

    await daiJoinContract.exit(await signer.getAddress(),
daiAmount)
}

```

DAI Savings Rate

MakerDAO offers the DAI Savings Rate (DSR), an interest rate paid to DAI holders who deposit into the DSR contract. The rate is funded by stability fees collected from Vault owners.

```

const DSR_MANAGER_ABI = [
'function join(address dst, uint256 wad) external',
'function exit(address dst, uint256 wad) external',
'function exitAll(address dst) external'
]

```

```

const POT_ABI = [
'function chi() external view returns (uint256)',
'function dsr() external view returns (uint256)',
'function pie(address) external view returns (uint256)'
]

```

```

async function depositToDSR(signer, daiAmount, dsrManager,
dai) {
    const daiContract = new ethers.Contract(dai, ERC20_ABI,
signer)
    const dsrContract = new ethers.Contract(dsrManager,
DSR_MANAGER_ABI, signer)

```

```

    await daiContract.approve(dsrManager, daiAmount)
    await dsrContract.join(await signer.getAddress(), daiAmount)
}

```

```

async function getDSRRate(provider, potAddress) {
    const pot = new ethers.Contract(potAddress, POT_ABI,

```


provider)

```
const dsr = await pot.dsr()
```

```
const dsrPerSecond = Number(dsr) / 1e27
```

```
const annualRate = (dsrPerSecond ** (365 * 24 * 60 * 60) -  
1) * 100
```

```
return annualRate
```

```
}
```

Liquity and LUSD

Liquity offers a different model with LUSD. It requires only 110% minimum collateralization ratio, making it more capital efficient than DAI. However, it only accepts ETH as collateral.

Liquity has no governance and charges one-time fees rather than ongoing interest. This makes it attractive for long-term positions.

```
const LIQUITY_ABI = {  
  borrowerOperations: [  
    'function openTrove(uint256 _maxFee, uint256  
_LUSDAmount, address _upperHint, address _lowerHint)  
external payable',  
    'function adjustTrove(uint256 _maxFee, uint256  
_collWithdrawal, uint256 _debtChange, bool isDebtIncrease,  
address _upperHint, address _lowerHint) external payable',  
    'function closeTrove() external'  
  ],  
  troveManager: [  
    'function getTroveDebt(address _borrower) external view  
returns (uint256)',  
    'function getTroveColl(address _borrower) external view  
returns (uint256)',
```

```
'function getNominalICR(address _borrower) external view  
returns (uint256)'  
]  
}
```

SECTION 4: ALGORITHMIC STABLECOINS

The Algorithmic Dream

Algorithmic stablecoins attempt to maintain peg through market incentives alone, without collateral backing. The appeal is capital efficiency. No locked collateral means all capital remains productive.

The design typically involves two tokens. A stablecoin and a governance/seigniorage token. When the stablecoin trades above peg, the system mints more stablecoins, increasing supply to lower price. When it trades below peg, the system contracts supply, often by offering bonds or burning the seigniorage token.

The Terra/Luna Collapse

Terra's UST was the largest algorithmic stablecoin, reaching \$18 billion before its catastrophic collapse in May 2022.

UST maintained its peg through arbitrage with LUNA. When UST traded below \$1, users could burn 1 UST for \$1 worth of LUNA, then sell LUNA for more than 1 UST, profiting and reducing UST supply. When UST traded above \$1, users could mint 1 UST by burning \$1 worth of LUNA.

The system collapsed when confidence failed. As UST depegged, users rushed to redeem for LUNA. This created massive LUNA selling pressure, crashing LUNA price. As

LUNA crashed, more LUNA was needed to redeem each UST, hyperinflating LUNA supply. The death spiral wiped out \$60 billion in value within days.

The lesson: algorithmic mechanisms require constant demand. When confidence breaks, reflexivity works in reverse with devastating speed.

Partial Collateralization (FRAX)

FRAX attempted a middle ground with fractional algorithmic design. Initially it used partial collateral (say 80%) plus algorithmic mechanisms (20%). The collateral ratio adjusted based on market conditions.

After Terra's collapse, FRAX shifted toward full collateralization. The algorithmic component introduced too much risk relative to its efficiency benefits.

The current FRAX model focuses on real-world assets and lending. It demonstrates how algorithmic ambitions have largely been abandoned in favor of more conservative designs.

Why Algorithmic Stablecoins Struggle

Pure algorithmic designs face fundamental challenges.

Reflexivity during depegs creates death spirals. The mechanism that should restore peg (burning governance tokens) actually accelerates decline as governance token holders flee.

No floor exists during panics. Collateralized stablecoins have liquidation mechanisms that sell collateral. Algorithmic stablecoins have only psychology and incentives which fail during panics.

Confidence is the only backing. Once confidence breaks, nothing prevents complete collapse.

The theoretical elegance of algorithmic stability has not survived contact with real markets.

SECTION 5: HYBRID AND EMERGING DESIGNS

Real-World Asset Backed

Some stablecoins use real-world assets beyond cash. MakerDAO now includes US treasury bonds in DAI backing. Specialized protocols tokenize specific asset classes.

Ondo Finance offers USDY backed by short-term US treasuries and bank deposits. Yield from these assets passes to token holders, creating an interest-bearing stablecoin.

Mountain Protocol's USDM similarly passes treasury yield to holders, offering 4-5% APY while maintaining dollar peg.

These designs blur the line between stablecoins and tokenized securities. Regulatory treatment varies by jurisdiction.

GHO (Aave Stablecoin)

Aave launched GHO, a stablecoin minted against Aave collateral positions. Users with Aave deposits can mint GHO at favorable rates, especially stkAAVE holders who receive discounts.

GHO introduces facilitators that can mint GHO according to preset rules. This modular design allows expansion to different minting mechanisms over time.

```
const GHO_ABI = [  
  'function mint(address to, uint256 amount) external',  
  'function burn(uint256 amount) external',  
  'function balanceOf(address account) external view returns  
(uint256)',  
  'function getFacilitatorBucket(address facilitator) external  
view returns (uint256 capacity, uint256 level)'  
]
```

GHO represents the trend of DeFi protocols launching native stablecoins to capture value from stablecoin usage within their ecosystems.

crvUSD (Curve Stablecoin)

Curve launched crvUSD with innovative liquidation mechanics. Rather than sudden liquidation events, crvUSD uses LLAMMA (Lending-Liquidating AMM Algorithm) to gradually convert collateral to stablecoins as prices decline.

This soft liquidation approach reduces MEV extraction and provides borrowers more time to manage positions during volatility.

SECTION 6: INTEGRATION PATTERNS

Multi-Stablecoin Support

Production applications should support multiple stablecoins for user convenience and risk distribution.

```
const STABLECOIN_CONFIG = {  
  ethereum: {  
    USDC: {  
      address:
```

```
'0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48',
  decimals: 6,
  type: 'fiat-backed'
},
USDT: {
  address:
'0xdAC17F958D2ee523a2206206994597C13D831ec7',
  decimals: 6,
  type: 'fiat-backed'
},
DAI: {
  address:
'0x6B175474E89094C44Da98b954EesdeCB5C69F33',
  decimals: 18,
  type: 'crypto-backed'
},
FRAX: {
  address:
'0x853d955aCEf822Db058eb8505911ED77F175b99e',
  decimals: 18,
  type: 'hybrid'
}
},
polygon: {
  USDC: {
    address: '0x3c499c542cEF5E3811e1192ce70d8cC03d5c3359',
    decimals: 6,
    type: 'fiat-backed'
  },
  USDT: {
    address:
'0xc2132D05D31c914a87C6611C10748AEb04B58e8F',
    decimals: 6,
    type: 'fiat-backed'
  }
}
}
```

```

function getStablecoin(chainId, symbol) {
  const chainConfig = {
    1: 'ethereum',
    137: 'polygon'
  }

  const chain = chainConfig[chainId]
  return STABLECOIN_CONFIG[chain]?.[symbol]
}

async function getStablecoinBalance(provider, chainId,
symbol, address) {
  const config = getStablecoin(chainId, symbol)
  if (!config) throw new Error(`Stablecoin ${symbol} not
configured for chain ${chainId}`)

  const contract = new ethers.Contract(
    config.address,
    ['function balanceOf(address) view returns (uint256)'],
    provider
  )

  const balance = await contract.balanceOf(address)
  return ethers.formatUnits(balance, config.decimals)
}

```

Handling Decimal Differences

Stablecoins use different decimal places. USDC and USDT use 6 decimals. DAI uses 18 decimals. Failing to account for this causes catastrophic errors.

```

function normalizeAmount(amount, fromDecimals,
toDecimals) {
  if (fromDecimals === toDecimals) {

```

```
return amount
}
```

```
if (fromDecimals > toDecimals) {
  const factor = BigInt(10) ** BigInt(fromDecimals -
toDecimals)
  return amount / factor
} else {
  const factor = BigInt(10) ** BigInt(toDecimals -
fromDecimals)
  return amount * factor
}
}
```

```
function parseStablecoinAmount(amountString, decimals) {
  return ethers.parseUnits(amountString, decimals)
}
```

```
function formatStablecoinAmount(amount, decimals) {
  return ethers.formatUnits(amount, decimals)
}
```

```
async function convertBetweenStablecoins(
  provider,
  fromConfig,
  toConfig,
  amount
) {
  const normalizedAmount = normalizeAmount(
    amount,
    fromConfig.decimals,
    toConfig.decimals
  )

  return normalizedAmount
}
```


Price Feed Integration

Although stablecoins target \$1, they can deviate. Applications handling significant value should verify prices.

```
const CHAINLINK_FEEDS = {
  'USDC/USD':
    '0x8fFfFd4AfB6115b954Bd326cbe7B4BA576818f6',
  'USDT/USD':
    '0x3E7d1eAB13ad0104d2750B8863b489D65364e32D',
  'DAI/USD':
    '0xAed0c38402a5d19df6E4c03F4E2DceD6e29c1ee9'
}

const CHAINLINK_ABI = [
  'function latestRoundData() external view returns (uint80
  roundId, int256 answer, uint256 startedAt, uint256
  updatedAt, uint80 answeredInRound)'
]

async function getStablecoinPrice(provider, symbol) {
  const feedAddress = CHAINLINK_FEEDS[`${symbol}/USD`]
  if (!feedAddress) throw new Error(`No price feed for
  ${symbol}`)

  const feed = new ethers.Contract(feedAddress,
  CHAINLINK_ABI, provider)
  const [, answer, , updatedAt] = await feed.latestRoundData()

  const price = Number(answer) / 1e8
  const lastUpdate = new Date(Number(updatedAt) * 1000)

  return {
    symbol,
    price,
    lastUpdate,
```

```
isStale: Date.now() - lastUpdate.getTime() > 3600000
}  
}
```

```
async function checkStablecoinPeg(provider, symbol,  
threshold = 0.02) {  
  const priceData = await getStablecoinPrice(provider,  
symbol)
```

```
  const deviation = Math.abs(priceData.price - 1)  
  const isDepegged = deviation > threshold
```

```
  return {  
    ...priceData,  
    deviation,  
    deviationPercent: deviation * 100,  
    isDepegged,  
    status: isDepegged ? 'WARNING' : 'OK'  
  }  
}
```

SECTION 7: DEPEGGING RISKS

Historical Depeg Events

Understanding past depegs helps prepare for future ones.

USDC March 2023. Circle held \$3.3 billion at Silicon Valley Bank when it failed. USDC briefly traded at \$0.87 before recovering when the Federal Reserve guaranteed deposits.

UST May 2022. Algorithmic failure led to complete collapse from \$1 to near zero.

DAI March 2020. During the COVID crash, DAI traded at \$1.10+ due to demand for stable assets and liquidation

mechanics.

USDT October 2018. Traded at \$0.92 on concerns about reserve backing, recovering over subsequent weeks.

Monitoring for Depegs

Real-time monitoring can provide early warning of instability.

```
class StablecoinMonitor {
  constructor(config) {
    this.provider = config.provider
    this.stablecoins = config.stablecoins
    this.alertThreshold = config.alertThreshold || 0.01
    this.criticalThreshold = config.criticalThreshold || 0.03
    this.onAlert = config.onAlert
  }

  async checkAllPegs() {
    const results = []

    for (const symbol of this.stablecoins) {
      const status = await checkStablecoinPeg(
        this.provider,
        symbol,
        this.alertThreshold
      )

      results.push(status)

      if (status.deviation > this.criticalThreshold) {
        this.onAlert({
          level: 'CRITICAL',
          symbol,
          deviation: status.deviationPercent,
          price: status.price
        })
      }
    }
  }
}
```

```

    })
    } else if (status.deviation > this.alertThreshold) {
    this.onAlert({
    level: 'WARNING',
    symbol,
    deviation: status.deviationPercent,
    price: status.price
    })
    }
    }

```

```

return results
}

```

```

async checkCurvePools() {
const pools = [
{ name: '3pool', address:
'0xbEbc44782C7dB0a1A60Cb6fe97d0b483032FF1C7' }
]

```

```

const CURVE_POOL_ABI = [
'function get_virtual_price() external view returns (uint256)',
'function balances(uint256) external view returns (uint256)'
]

```

```

const results = []

```

```

for (const pool of pools) {
const contract = new ethers.Contract(
pool.address,
CURVE_POOL_ABI,
this.provider
)

```

```

const balances = await Promise.all([
contract.balances(0),
contract.balances(1),

```

```
contract.balances(2)
])
```

```
const total = balances.reduce((a, b) => a + b, 0n)
const imbalances = balances.map(b => {
  const share = Number(b) / Number(total)
  const deviation = Math.abs(share - 0.333)
  return deviation
})
```

```
const maxImbalance = Math.max(...imbalances)
```

```
results.push({
  pool: pool.name,
  imbalance: maxImbalance,
  isImbalanced: maxImbalance > 0.05
})
}
```

```
return results
}
```

```
startMonitoring(intervalMs = 60000) {
  this.interval = setInterval(() => this.checkAllPegs(),
intervalMs)
  this.checkAllPegs()
}
```

```
stopMonitoring() {
  if (this.interval) {
    clearInterval(this.interval)
  }
}
}
```

Protecting Against Depegs

Applications can implement protective measures.

```
async function safeStablecoinTransfer(
  signer,
  stablecoinConfig,
  to,
  amount,
  maxDeviation = 0.02
) {
  const pegStatus = await checkStablecoinPeg(
    signer.provider,
    stablecoinConfig.symbol,
    maxDeviation
  )

  if (pegStatus.isDepegged) {
    throw new Error(
      `${stablecoinConfig.symbol} is depegged: ${pegStatus.price}.
    +
    `Deviation: ${pegStatus.deviationPercent.toFixed(2)}%`
    )
  }

  const contract = new ethers.Contract(
    stablecoinConfig.address,
    ['function transfer(address, uint256) returns (bool)'],
    signer
  )

  return contract.transfer(to, amount)
}

function diversifyStablecoinHoldings(totalAmount, allocation)
{
  const holdings = {}
```

```

for (const [symbol, percentage] of Object.entries(allocation))
{
  holdings[symbol] = totalAmount * percentage / 100
}

return holdings
}

```

SECTION 8: STABLECOIN SWAPS

Curve Finance for Stablecoin Swaps

Curve specializes in stablecoin and pegged asset swaps with minimal slippage.

```

const CURVE_3POOL =
'0xbEbc44782C7dB0a1A60Cb6fe97d0b483032FF1C7'

const CURVE_POOL_ABI = [
  'function exchange(int128 i, int128 j, uint256 dx, uint256
min_dy) external returns (uint256)',
  'function get_dy(int128 i, int128 j, uint256 dx) external view
returns (uint256)',
  'function coins(uint256) external view returns (address)'
]

async function getCurveSwapQuote(provider, poolAddress,
fromIndex, toIndex, amount) {
  const pool = new ethers.Contract(poolAddress,
CURVE_POOL_ABI, provider)

  const expectedOut = await pool.get_dy(fromIndex, toIndex,
amount)

  return expectedOut
}

```

```

async function executeCurveSwap(
  signer,
  poolAddress,
  fromIndex,
  toIndex,
  amount,
  minOut,
  fromTokenAddress
) {
  const token = new ethers.Contract(
    fromTokenAddress,
    ['function approve(address, uint256) returns (bool)'],
    signer
  )

  await token.approve(poolAddress, amount)

  const pool = new ethers.Contract(poolAddress,
    CURVE_POOL_ABI, signer)

  const tx = await pool.exchange(fromIndex, toIndex, amount,
    minOut)
  return tx.wait()
}

```

Aggregator Integration for Best Rates

For larger swaps, aggregators find optimal routing.

```

async function findBestStablecoinSwap(
  fromToken,
  toToken,
  amount,
  chainId
) {
  const quotes = await Promise.allSettled([

```



```

get1inchQuote(chainId, fromToken, toToken, amount),
get0xQuote(chainId, fromToken, toToken, amount),
getCurveDirectQuote(fromToken, toToken, amount)
])

```

```

const validQuotes = quotes
.filter(q => q.status === 'fulfilled')
.map((q, i) => ({
source: ['1inch', '0x', 'Curve'][i],
quote: q.value
}))

```

```

if (validQuotes.length === 0) {
throw new Error('No valid quotes found')
}

```

```

validQuotes.sort((a, b) => {
const aOut = BigInt(a.quote.toAmount || a.quote.buyAmount
|| a.quote)
const bOut = BigInt(b.quote.toAmount ||
b.quote.buyAmount || b.quote)
return bOut > aOut ? 1 : -1
})

```

```

return validQuotes[0]
}

```

SECTION 9: BUILDING WITH STABLECOINS

Payment Processing

Stablecoins excel for payment processing due to stable value and fast settlement.

```

class StablecoinPaymentProcessor {
constructor(config) {

```

```
this.provider = config.provider
this.acceptedStablecoins = config.acceptedStablecoins
this.receiverAddress = config.receiverAddress
this.onPaymentReceived = config.onPaymentReceived
}
```

```
generatePaymentRequest(amountUSD, orderId, expiresIn =
3600) {
  const request = {
    orderId,
    amountUSD,
    acceptedTokens: this.acceptedStablecoins.map(s => ({
      symbol: s.symbol,
      address: s.address,
      amount: parseStablecoinAmount(amountUSD.toString(),
s.decimals).toString()
    })),
    receiver: this.receiverAddress,
    expiresAt: Date.now() + expiresIn * 1000
  }
}
```

```
return request
}
```

```
async verifyPayment(request, txHash) {
  const receipt = await
this.provider.getTransactionReceipt(txHash)
```

```
  if (!receipt || receipt.status !== 1) {
    return { verified: false, reason: 'Transaction failed or not
found' }
  }
```

```
  const transferTopic =
ethers.id('Transfer(address,address,uint256)')
```

```
  for (const log of receipt.logs) {
```

```
if (log.topics[0] !== transferTopic) continue
```

```
const tokenConfig = this.acceptedStablecoins.find(  
  s => s.address.toLowerCase() ===  
  log.address.toLowerCase()  
)
```

```
if (!tokenConfig) continue
```

```
const toAddress = '0x' + log.topics[2].slice(26)  
if (toAddress.toLowerCase() !==  
this.receiverAddress.toLowerCase()) continue
```

```
const amount = BigInt(log.data)  
const expectedAmount = parseStablecoinAmount(  
  request.amountUSD.toString(),  
  tokenConfig.decimals  
)
```

```
if (amount >= expectedAmount) {  
  return {  
    verified: true,  
    token: tokenConfig.symbol,  
    amount: formatStablecoinAmount(amount,  
    tokenConfig.decimals),  
    txHash  
  }  
}  
}
```

```
return { verified: false, reason: 'Payment not found in  
transaction' }  
}
```

```
async watchForPayment(request, callback, pollInterval =  
5000) {  
  const startBlock = await this.provider.getBlockNumber()
```

```

const checkPayment = async () => {
  const currentBlock = await this.provider.getBlockNumber()

  for (const token of this.acceptedStablecoins) {
    const contract = new ethers.Contract(
      token.address,
      ['event Transfer(address indexed from, address indexed to,
uint256 value)'],
      this.provider
    )

    const filter = contract.filters.Transfer(null,
this.receiverAddress)
    const events = await contract.queryFilter(filter, startBlock,
currentBlock)

    for (const event of events) {
      const expectedAmount = parseStablecoinAmount(
request.amountUSD.toString(),
token.decimals
      )

      if (event.args.value >= expectedAmount) {
        callback({
          verified: true,
          token: token.symbol,
          amount: formatStablecoinAmount(event.args.value,
token.decimals),
          txHash: event.transactionHash,
          from: event.args.from
        })
        return true
      }
    }
  }
}

```

```

return false
}

const interval = setInterval(async () => {
  const found = await checkPayment()
  if (found) {
    clearInterval(interval)
  }
}, pollInterval)

return () => clearInterval(interval)
}
}

```

Invoice Generation

Generate invoices with multiple payment options.

```

function generateStablecoinInvoice(params) {
  const {
    invoiceId,
    amountUSD,
    description,
    receiverAddress,
    acceptedStablecoins,
    expiresAt,
    metadata
  } = params

  const paymentOptions = acceptedStablecoins.map(token
=> {
    const amount =
parseStablecoinAmount(amountUSD.toString(),
token.decimals)

    return {

```

```

token: token.symbol,
chain: token.chain,
address: token.address,
amount: amount.toString(),
amountFormatted: amountUSD.toFixed(2)
}
})

```

```

const invoice = {
  id: invoiceId,
  version: '1.0',
  created: new Date().toISOString(),
  expires: expiresAt ? new Date(expiresAt).toISOString() : null,
  amount: {
    value: amountUSD,
    currency: 'USD'
  },
  description,
  receiver: receiverAddress,
  paymentOptions,
  metadata,
  status: 'pending'
}

return invoice
}

```

CHAPTER SUMMARY

Stablecoins provide price stability essential for DeFi's lending, trading, and payment functions. Different designs offer different tradeoffs between decentralization, capital efficiency, and risk.

Fiat-collateralized stablecoins like USDC and USDT offer simplicity and reliability but require trusting centralized

issuers. Crypto-collateralized stablecoins like DAI achieve decentralization through overcollateralization. Algorithmic designs have largely failed, with Terra's collapse demonstrating the fragility of pure algorithmic mechanisms.

Integration requires handling decimal differences between tokens, monitoring peg stability, and supporting multiple stablecoins for user convenience and risk distribution.

Depegging risks are real. Historical events show even major stablecoins can temporarily lose their peg. Monitoring and protective measures help applications respond appropriately.

Curve Finance provides optimal routing for stablecoin swaps with minimal slippage. Aggregators help find best rates across venues.

Building with stablecoins enables payment processing, invoicing, and commerce applications that benefit from cryptocurrency rails while avoiding volatility.

The next chapter explores tokenomics design, examining how to create sustainable token economic models for new protocols.

CHAPTER 6: TOKENOMICS DESIGN

Tokenomics determines whether a project thrives or dies. Brilliant technology with poor token economics fails. Mediocre technology with well-designed tokenomics can succeed. The token is how value flows through your protocol. Design it wrong and value leaks to speculators, early insiders, or simply evaporates.

This chapter covers the fundamental principles of token economic design: supply mechanics, inflation and deflation dynamics, burning mechanisms, and the critical distinction between utility and governance functions.

SECTION 1: TOKEN ECONOMICS FUNDAMENTALS

What Is Tokenomics

Tokenomics combines "token" and "economics" to describe the economic design of cryptocurrency tokens. It encompasses supply and demand dynamics, distribution mechanisms, utility functions, and incentive structures.

Good tokenomics aligns incentives. Users benefit from using the protocol. Token holders benefit from protocol success. Developers benefit from building. All parties pull in the same direction.

Bad tokenomics creates misalignment. Insiders extract value from users. Short-term speculators profit at long-term holders' expense. Usage does not translate to token value.

The Value Capture Problem

Many protocols struggle to capture value in their token. Users benefit from the protocol but the token provides no advantage. Uniswap processes billions in volume but UNI holders receive none of it. Ethereum processes all DeFi transactions and ETH captures that value through gas fees.

Value capture mechanisms include fee switches that direct revenue to token holders, staking requirements that create demand, burning mechanisms that reduce supply, and governance rights that control treasury.

The fundamental question: why would someone buy and hold this token? If the answer is "speculation on future value," the tokenomics are weak. If the answer is "because holding provides concrete benefits," the tokenomics are strong.

Supply and Demand Framework

Token price ultimately reflects supply and demand.

Demand comes from utility (needing tokens to use the protocol), speculation (betting on price appreciation), and yield (earning returns by holding).

Supply comes from initial distribution, ongoing emissions, and unlocks from vesting schedules.

$\text{Price} = \text{Demand} / \text{Supply}$

Sustainable tokenomics increases demand faster than supply grows. Unsustainable tokenomics does the opposite.

SECTION 2: SUPPLY MECHANICS

Fixed vs Unlimited Supply

Fixed supply tokens have a maximum cap. Bitcoin's 21 million limit is the canonical example. Scarcity drives value if demand exists.

Unlimited supply tokens can mint forever. Ethereum pre-merge had unlimited supply. Dogecoin inflates forever. Unlimited supply works if demand grows faster than supply.

Neither is inherently better. Fixed supply creates scarcity but limits flexibility. Unlimited supply enables ongoing incentives but requires careful emission management.

Total Supply vs Circulating Supply

Total supply includes all tokens that exist or will exist according to the protocol rules.

Circulating supply includes only tokens currently tradeable on the market.

These can differ dramatically. A token with 1 billion total supply might have only 100 million circulating if 900 million are locked in vesting contracts.

Market cap uses circulating supply. Fully diluted valuation uses total supply. The difference reveals future dilution.

```
def calculate_dilution_schedule(total_supply,
current_circulating, vesting_releases):
    schedule = []
    running_circulating = current_circulating

    for release in vesting_releases:
        running_circulating += release['amount']
        dilution_from_start = (running_circulating /
current_circulating - 1) * 100

        schedule.append({
            'date': release['date'],
            'new_circulating': running_circulating,
            'percent_of_total': running_circulating / total_supply * 100,
            'dilution_percent': dilution_from_start
        })

    return schedule
```

Emission Schedules

Emission schedules determine how new tokens enter circulation beyond initial distribution.

Linear emissions release the same amount each period. Simple but creates constant sell pressure.

Halving emissions cut the rate periodically. Bitcoin halves every four years. Creates decreasing inflation rate over time.

Curve emissions decrease smoothly rather than in discrete steps. Many DeFi protocols use this approach.

```
class EmissionSchedule:
    def __init__(self, initial_rate, schedule_type, params):
        self.initial_rate = initial_rate
        self.schedule_type = schedule_type
        self.params = params

    def get_emission_at_year(self, year):
        if self.schedule_type == 'linear':
            return self.initial_rate

        elif self.schedule_type == 'halving':
            halvings = year // self.params['halving_period']
            return self.initial_rate / (2 ** halvings)

        elif self.schedule_type == 'curve':
            decay_rate = self.params['decay_rate']
            return self.initial_rate * (decay_rate ** year)

        elif self.schedule_type == 'stepped':
            for threshold, rate in self.params['steps']:
                if year < threshold:
                    return rate
            return self.params['final_rate']

    def calculate_total_emissions(self, years):
        total = 0
        for year in range(years):
            total += self.get_emission_at_year(year)
```

```
return total
```

```
def project_supply(self, initial_supply, years):
```

```
    supply = initial_supply
```

```
    projection = [{'year': 0, 'supply': supply}]
```

```
    for year in range(1, years + 1):
```

```
        supply += self.get_emission_at_year(year)
```

```
        projection.append({'year': year, 'supply': supply})
```

```
    return projection
```

SECTION 3: INFLATION AND DEFLATION

Understanding Token Inflation

Inflation occurs when supply grows faster than demand. More tokens chase the same amount of value, diluting each token's worth.

$$\text{Inflation rate} = (\text{New tokens minted per year} / \text{Circulating supply}) * 100$$

High inflation is not inherently bad if demand grows faster. Low inflation is not inherently good if demand is falling.

The key metric is real return: nominal yield minus inflation rate. Earning 20% staking rewards with 25% inflation means losing 5% in real terms.

```
def calculate_real_return(nominal_yield, inflation_rate):
```

```
    real_return = ((1 + nominal_yield/100) / (1 +  
inflation_rate/100) - 1) * 100
```

```
    return real_return
```

```
def analyze_staking_economics(  
    staking_yield,
```

```

inflation_rate,
staking_ratio
):
    non_staker_dilution = inflation_rate

    staker_nominal = staking_yield
    staker_real = calculate_real_return(staking_yield,
inflation_rate)

    inflation_to_stakers = inflation_rate * staking_ratio
    inflation_to_others = inflation_rate * (1 - staking_ratio)

    return {
'staker_nominal_yield': staker_nominal,
'staker_real_yield': staker_real,
'non_staker_dilution': non_staker_dilution,
'staking_captures_inflation': inflation_to_stakers
}

```

Deflationary Mechanics

Deflation occurs when supply decreases. Burning tokens, buybacks, or simple supply caps with lost tokens all contribute.

Ethereum became deflationary after the merge. Base fees are burned while issuance to validators is reduced. During high activity, more ETH burns than is issued.

```

def model_deflationary_token(
    initial_supply,
    annual_burn_rate,
    annual_emission,
    years
):
    supply = initial_supply

```

```

history = [{'year': 0, 'supply': supply}]

for year in range(1, years + 1):
    burned = supply * annual_burn_rate
    minted = annual_emission

    net_change = minted - burned
    supply = max(0, supply + net_change)

    history.append({
        'year': year,
        'supply': supply,
        'burned': burned,
        'minted': minted,
        'net_change': net_change,
        'is_deflationary': net_change < 0
    })

return history

```

Equilibrium Dynamics

Well-designed tokenomics reach equilibrium where supply and demand pressures balance.

For inflationary tokens, equilibrium requires demand growth matching or exceeding inflation. Staking rewards, utility requirements, and ecosystem growth drive demand.

For deflationary tokens, equilibrium requires sustainable burn sources. Transaction fees, buybacks, or usage-based burns must persist as supply decreases.

```

def find_equilibrium_price(
    current_price,
    current_supply,

```

```

demand_growth_rate,
supply_growth_rate,
periods
):
    price = current_price
    supply = current_supply
    demand = current_price * current_supply

    history = []

    for period in range(periods):
        demand *= (1 + demand_growth_rate)
        supply *= (1 + supply_growth_rate)

        price = demand / supply

        history.append({
            'period': period,
            'price': price,
            'supply': supply,
            'demand': demand,
            'market_cap': price * supply
        })

    return history

```

SECTION 4: BURNING MECHANISMS

Transaction Fee Burns

Burning a portion of transaction fees reduces supply with every protocol interaction. Ethereum's EIP-1559 burns base fees, making ETH deflationary during high usage.

```

pragma solidity ^0.8.24;

contract TokenWithBurn {

```

```
string public name = "Burnable Token";  
string public symbol = "BURN";  
uint8 public decimals = 18;  
uint256 public totalSupply;
```

```
uint256 public burnRate = 100;  
uint256 public constant RATE_DENOMINATOR = 10000;
```

```
mapping(address => uint256) public balanceOf;  
mapping(address => mapping(address => uint256))  
public allowance;
```

```
uint256 public totalBurned;
```

```
event Transfer(address indexed from, address indexed to,  
uint256 value);  
event Burn(address indexed from, uint256 value);
```

```
constructor(uint256 initialSupply) {  
    totalSupply = initialSupply;  
    balanceOf[msg.sender] = initialSupply;  
}
```

```
function transfer(address to, uint256 value) external returns  
(bool) {  
    return _transfer(msg.sender, to, value);  
}
```

```
function transferFrom(address from, address to, uint256  
value) external returns (bool) {  
    require(allowance[from][msg.sender] >= value,  
"Insufficient allowance");  
    allowance[from][msg.sender] -= value;  
    return _transfer(from, to, value);  
}
```

```
function _transfer(address from, address to, uint256 value)
```



```

internal returns (bool) {
    require(balanceOf[from] >= value, "Insufficient balance");

    uint256 burnAmount = (value * burnRate) /
RATE_DENOMINATOR;
    uint256 transferAmount = value - burnAmount;

    balanceOf[from] -= value;
    balanceOf[to] += transferAmount;

    if (burnAmount > 0) {
        totalSupply -= burnAmount;
        totalBurned += burnAmount;
        emit Burn(from, burnAmount);
    }

    emit Transfer(from, to, transferAmount);
    return true;
}

function approve(address spender, uint256 value) external
returns (bool) {
    allowance[msg.sender][spender] = value;
    return true;
}
}

```

Protocol Revenue Burns

Instead of distributing revenue to token holders, some protocols burn tokens purchased with revenue.

```

pragma solidity ^0.8.24;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/

```

```
ERC20Burnable.sol";
```

```
interface ISwapRouter {  
    function swapExactTokensForTokens(  
        uint256 amountIn,  
        uint256 amountOutMin,  
        address[] calldata path,  
        address to,  
        uint256 deadline  
    ) external returns (uint256[] memory amounts);  
}
```

```
contract BuybackAndBurn {  
    IERC20 public revenueToken;  
    ERC20Burnable public protocolToken;  
    ISwapRouter public router;
```

```
    address public owner;  
    uint256 public totalBurned;
```

```
    address[] public swapPath;
```

```
    event BuybackExecuted(uint256 revenueSpent, uint256  
tokensBurned);
```

```
    constructor(  
        address _revenueToken,  
        address _protocolToken,  
        address _router  
    ) {
```

```
        revenueToken = IERC20(_revenueToken);  
        protocolToken = ERC20Burnable(_protocolToken);  
        router = ISwapRouter(_router);  
        owner = msg.sender;
```

```
        swapPath = new address[](2);  
        swapPath[0] = _revenueToken;
```

```

swapPath[1] = _protocolToken;

revenueToken.approve(_router, type(uint256).max);
}

function executeBuybackAndBurn(uint256 minTokensOut)
external {
    require(msg.sender == owner, "Only owner");

    uint256 revenueBalance =
revenueToken.balanceOf(address(this));
    require(revenueBalance > 0, "No revenue to burn");

    uint256[] memory amounts =
router.swapExactTokensForTokens(
    revenueBalance,
    minTokensOut,
    swapPath,
    address(this),
    block.timestamp
);

    uint256 tokensBought = amounts[amounts.length - 1];

    protocolToken.burn(tokensBought);
    totalBurned += tokensBought;

    emit BuybackExecuted(revenueBalance, tokensBought);
}

function pendingRevenue() external view returns (uint256) {
    return revenueToken.balanceOf(address(this));
}
}

```

Voluntary Burns

Some mechanisms incentivize users to voluntarily burn tokens for benefits.

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";  
import "@openzeppelin/contracts/token/ERC20/extensions/  
ERC20Burnable.sol";
```

```
contract BurnForBenefits {  
    ERC20Burnable public token;
```

```
    mapping(address => uint256) public burnedByUser;  
    mapping(address => uint8) public userTier;
```

```
    uint256[] public tierThresholds = [  
        1000 * 1e18,  
        10000 * 1e18,  
        100000 * 1e18  
    ];
```

```
    uint256[] public tierDiscounts = [10, 25, 50];
```

```
    uint256 public totalBurnedForTiers;
```

```
    event TierUpgrade(address indexed user, uint8 newTier,  
        uint256 totalBurned);
```

```
    constructor(address _token) {  
        token = ERC20Burnable(_token);  
    }
```

```
    function burnForTier(uint256 amount) external {  
        require(amount > 0, "Amount must be positive");
```

```
        token.burnFrom(msg.sender, amount);
```

```
        burnedByUser[msg.sender] += amount;
```

```
totalBurnedForTiers += amount;
```

```
uint8 newTier = calculateTier(burnedByUser[msg.sender]);
```

```
if (newTier > userTier[msg.sender]) {  
    userTier[msg.sender] = newTier;  
    emit TierUpgrade(msg.sender, newTier,  
burnedByUser[msg.sender]);  
}  
}
```

```
function calculateTier(uint256 totalBurned) public view  
returns (uint8) {  
    for (uint8 i = uint8(tierThresholds.length); i > 0; i--) {  
        if (totalBurned >= tierThresholds[i - 1]) {  
            return i;  
        }  
    }  
    return 0;  
}
```

```
function getDiscount(address user) external view returns  
(uint256) {  
    uint8 tier = userTier[user];  
    if (tier == 0) return 0;  
    return tierDiscounts[tier - 1];  
}
```

```
function burnedToNextTier(address user) external view  
returns (uint256) {  
    uint8 currentTier = userTier[user];  
    if (currentTier >= tierThresholds.length) return 0;  
  
    return tierThresholds[currentTier] - burnedByUser[user];  
}  
}
```

SECTION 5: UTILITY VS GOVERNANCE TOKENS

Pure Governance Tokens

Governance tokens grant voting rights over protocol decisions. Holders can propose and vote on parameter changes, treasury allocation, and protocol upgrades.

UNI is the canonical example. It controls the Uniswap protocol but does not receive any protocol fees. The fee switch has never been activated. UNI value derives entirely from potential future value capture and speculation.

Pure governance tokens struggle to maintain value because governance rights have limited intrinsic worth. Why buy the right to vote on a protocol you do not use intensively?

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";  
import "@openzeppelin/contracts/token/ERC20/extensions/  
ERC20Votes.sol";
```

```
contract GovernanceToken is ERC20, ERC20Votes {  
    constructor(uint256 initialSupply)  
        ERC20("Governance Token", "GOV")  
        ERC20Permit("Governance Token")  
    {  
        _mint(msg.sender, initialSupply);  
    }  
}
```

```
function _update(address from, address to, uint256 amount)  
    internal  
    override(ERC20, ERC20Votes)  
    {  
        super._update(from, to, amount);  
    }  
}
```

```
}  
  
function nonces(address owner)  
public  
view  
override(ERC20Permit, Nonces)  
returns (uint256)  
{  
    return super.nonces(owner);  
}  
}
```

Pure Utility Tokens

Utility tokens provide functional benefits within a protocol. You need them to use the service.

Filecoin's FIL is needed to pay for storage. Chainlink's LINK is used to pay oracle operators. BNB provides fee discounts on Binance.

Pure utility tokens have clearer value propositions. Demand ties directly to usage. But utility alone may not justify high valuations if the utility is narrow.

Hybrid Models

Most successful tokens combine utility and governance.

AAVE can be staked in the Safety Module for protocol insurance and rewards. It also governs Aave protocol parameters. Utility plus governance.

CRV can be locked for veCRV, earning trading fees and boosting LP rewards. It also governs Curve gauge weights. Utility, yield, and governance combined.

```

class TokenomicsAnalysis:
    def __init__(self, token):
        self.token = token

    def analyze_value_accrual(self):
        mechanisms = {
            'fee_sharing': self.token.get('fee_share_percent', 0),
            'staking_yield': self.token.get('staking_yield', 0),
            'burn_rate': self.token.get('annual_burn_rate', 0),
            'utility_discount': self.token.get('utility_discount', 0),
            'governance_value': self.token.get('treasury_controlled', 0)
        }

        total_value_capture = sum([
            mechanisms['fee_sharing'],
            mechanisms['staking_yield'],
            mechanisms['burn_rate'],
            mechanisms['utility_discount'] * 0.1
        ])

        return {
            'mechanisms': mechanisms,
            'total_value_capture_score': total_value_capture,
            'primary_value_driver': max(mechanisms,
            key = mechanisms.get)
        }

```

Designing for Value Capture

Effective tokenomics create multiple demand sinks.

Fee capture: Revenue flows to token holders through staking dividends or buyback/burn.

Utility requirements: Using the protocol requires holding or

spending tokens.

Staking requirements: Participating as a service provider requires staking tokens at risk.

Governance power: Meaningful decisions require token voting, creating demand from those who want influence.

```
def design_value_capture(protocol_params):
    revenue = protocol_params['annual_revenue']
    market_cap = protocol_params['market_cap']
    staking_ratio = protocol_params['target_staking_ratio']

    fee_to_stakers = revenue * 0.5
    staked_value = market_cap * staking_ratio
    staking_apy = (fee_to_stakers / staked_value) * 100

    burn_allocation = revenue * 0.2
    annual_burn_percent = (burn_allocation / market_cap) * 100

    treasury_allocation = revenue * 0.3

    return {
        'fee_sharing': {
            'allocation_percent': 50,
            'staking_apy': staking_apy,
            'notes': 'Distributed to stakers proportionally'
        },
        'buyback_burn': {
            'allocation_percent': 20,
            'annual_supply_reduction': annual_burn_percent,
            'notes': 'Protocol buys and burns tokens'
        },
        'treasury': {
            'allocation_percent': 30,
            'annual_addition': treasury_allocation,
            'notes': 'Controlled by governance for growth'
```

```
}  
}
```

SECTION 6: STAKING ECONOMICS

Staking as a Supply Sink

Staking locks tokens, removing them from circulating supply. This reduces selling pressure and creates alignment between holders and protocol.

The key variables are staking yield, lock duration, and unstaking period. Higher yields and shorter locks attract more stakers but may not create durable alignment.

```
def model_staking_equilibrium(  
    total_supply,  
    staking_yield,  
    opportunity_cost,  
    risk_premium  
):  
    required_yield = opportunity_cost + risk_premium  
  
    if staking_yield <= required_yield:  
        equilibrium_staking = 0.1  
    else:  
        yield_spread = staking_yield - required_yield  
        equilibrium_staking = min(0.9, 0.3 + yield_spread * 2)  
  
    staked_supply = total_supply * equilibrium_staking  
    circulating_supply = total_supply * (1 - equilibrium_staking)  
  
    return {  
        'equilibrium_staking_ratio': equilibrium_staking,  
        'staked_supply': staked_supply,  
        'circulating_supply': circulating_supply,
```

```
'effective_supply_reduction': equilibrium_staking * 100  
}
```

Slashing Mechanics

Proof-of-stake networks and some DeFi protocols implement slashing to penalize misbehavior.

```
pragma solidity ^0.8.24;
```

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";  
import "@openzeppelin/contracts/token/ERC20/utils/  
SafeERC20.sol";
```

```
contract SlashableStaking {  
    using SafeERC20 for IERC20;
```

```
    IERC20 public stakingToken;
```

```
    mapping(address => uint256) public stakedBalance;  
    mapping(address => uint256) public slashedAmount;
```

```
    uint256 public totalStaked;  
    uint256 public totalSlashed;
```

```
    address public slasher;  
    address public treasury;
```

```
    uint256 public maxSlashPercent = 1000;  
    uint256 public constant PERCENT_DENOMINATOR = 10000;
```

```
    event Staked(address indexed user, uint256 amount);  
    event Unstaked(address indexed user, uint256 amount);  
    event Slashed(address indexed user, uint256 amount, string  
reason);
```

```
    constructor(address _stakingToken, address _slasher, address
```

```
_treasury) {  
    stakingToken = IERC20(_stakingToken);  
    slasher = _slasher;  
    treasury = _treasury;  
}
```

```
function stake(uint256 amount) external {  
    require(amount > 0, "Cannot stake 0");
```

```
    stakingToken.safeTransferFrom(msg.sender, address(this),  
    amount);
```

```
    stakedBalance[msg.sender] += amount;  
    totalStaked += amount;
```

```
    emit Staked(msg.sender, amount);  
}
```

```
function unstake(uint256 amount) external {  
    require(stakedBalance[msg.sender] >= amount, "Insufficient  
    stake");
```

```
    stakedBalance[msg.sender] -= amount;  
    totalStaked -= amount;
```

```
    stakingToken.safeTransfer(msg.sender, amount);
```

```
    emit Unstaked(msg.sender, amount);  
}
```

```
function slash(address user, uint256 percent, string calldata  
reason) external {  
    require(msg.sender == slasher, "Only slasher");  
    require(percent <= maxSlashPercent, "Exceeds max slash");
```

```
    uint256 slashAmount = (stakedBalance[user] * percent) /  
    PERCENT_DENOMINATOR;
```

```

stakedBalance[user] -= slashAmount;
totalStaked -= slashAmount;
totalSlashed += slashAmount;
slashedAmount[user] += slashAmount;

stakingToken.safeTransfer(treasury, slashAmount);

emit Slashed(user, slashAmount, reason);
}

function effectiveStake(address user) external view returns
(uint256) {
    return stakedBalance[user];
}
}

```

Liquid Staking Tokens

Liquid staking solves the liquidity problem of staked tokens. Instead of locking tokens directly, users receive a liquid derivative representing their staked position.

The derivative can be traded, used as collateral, or deployed in DeFi while the underlying tokens remain staked earning rewards.

This creates composability but also introduces additional smart contract risk and potential depeg risk for the liquid token.

SECTION 7: MODELING TOKEN ECONOMICS

Supply and Demand Simulation

Model how token economics play out over time.

```
import random
```

```
class TokenEconomicsSimulator:
```

```
    def __init__(self, config):
```

```
        self.total_supply = config['initial_supply']
```

```
        self.circulating_supply = config['initial_circulating']
```

```
        self.price = config['initial_price']
```

```
        self.treasury = config.get('initial_treasury', 0)
```

```
        self.emission_rate = config['annual_emission_rate']
```

```
        self.burn_rate = config['annual_burn_rate']
```

```
        self.staking_ratio = config['initial_staking_ratio']
```

```
        self.staking_yield = config['staking_yield']
```

```
        self.demand_growth = config.get('demand_growth_rate', 0.1)
```

```
        self.demand_volatility = config.get('demand_volatility', 0.2)
```

```
        self.history = []
```

```
    def simulate_period(self):
```

```
        new_emissions = self.total_supply * self.emission_rate / 12
```

```
        self.total_supply += new_emissions
```

```
        self.circulating_supply += new_emissions * 0.7
```

```
        burned = self.circulating_supply * self.burn_rate / 12
```

```
        self.total_supply -= burned
```

```
        self.circulating_supply -= burned
```

```
        demand_change = self.demand_growth / 12
```

```
        demand_change += random.gauss(0, self.demand_volatility  
/ 12)
```

```
        effective_circulating = self.circulating_supply * (1 -  
self.staking_ratio)
```

```
        supply_change = (new_emissions - burned) /  
self.circulating_supply
```

```
price_change = demand_change - supply_change
self.price *= (1 + price_change)
self.price = max(0.001, self.price)
```

```
target_staking = 0.3 + (self.staking_yield - 0.05) * 2
self.staking_ratio = self.staking_ratio * 0.9 + target_staking *
0.1
self.staking_ratio = max(0.1, min(0.9, self.staking_ratio))
```

```
self.history.append({
'total_supply': self.total_supply,
'circulating_supply': self.circulating_supply,
'price': self.price,
'market_cap': self.price * self.circulating_supply,
'fdv': self.price * self.total_supply,
'staking_ratio': self.staking_ratio,
'emissions': new_emissions,
'burned': burned
})
```

```
def run_simulation(self, months):
for _ in range(months):
self.simulate_period()
```

```
return self.history
```

```
def analyze_results(self):
if not self.history:
return None
```

```
initial = self.history[0]
final = self.history[-1]
```

```
return {
'supply_change': (final['total_supply'] / initial['total_supply'] -
1) * 100,
```

```

'price_change': (final['price'] / initial['price'] - 1) * 100,
'market_cap_change': (final['market_cap'] /
initial['market_cap'] - 1) * 100,
'final_staking_ratio': final['staking_ratio'],
'total_burned': sum(h['burned'] for h in self.history),
'total_emitted': sum(h['emissions'] for h in self.history)
}

```

Scenario Analysis

Test tokenomics under different market conditions.

```

def run_scenario_analysis(base_config, scenarios):
    results = {}

    for scenario_name, scenario_params in scenarios.items():
        config = {**base_config, **scenario_params}
        simulator = TokenEconomicsSimulator(config)
        simulator.run_simulation(36)
        results[scenario_name] = simulator.analyze_results()

    return results

base_config = {
    'initial_supply': 1_000_000_000,
    'initial_circulating': 200_000_000,
    'initial_price': 1.0,
    'annual_emission_rate': 0.10,
    'annual_burn_rate': 0.02,
    'initial_staking_ratio': 0.3,
    'staking_yield': 0.08
}

scenarios = {
    'bull_market': {
        'demand_growth_rate': 0.30,

```



```
'demand_volatility': 0.3
},
'bear_market': {
'demand_growth_rate': -0.10,
'demand_volatility': 0.4
},
'high_inflation': {
'annual_emission_rate': 0.25,
'demand_growth_rate': 0.15
},
'deflationary': {
'annual_emission_rate': 0.02,
'annual_burn_rate': 0.05,
'demand_growth_rate': 0.10
}
}
```

SECTION 8: PRACTICAL TOKENOMICS DESIGN

Design Checklist

When designing tokenomics, systematically address each element.

Supply: Total supply, initial circulating, emission schedule, vesting periods.

Demand: Utility requirements, staking incentives, governance rights, speculation drivers.

Distribution: Team allocation, investor allocation, community allocation, treasury.

Value capture: Fee mechanisms, burn mechanics, staking rewards, buybacks.

Sustainability: Can emissions continue indefinitely? Do incentives remain attractive as the protocol matures?

Common Mistakes

High emissions without utility. Tokens distributed as rewards have no intrinsic demand. Recipients sell immediately, crashing price.

Overallocation to insiders. Heavy team and investor allocations create large unlocks that crush price when vesting ends.

No value capture. The protocol succeeds but the token does not benefit. Users capture all value.

Complexity without purpose. Convoluted mechanisms that confuse users without adding real value.

Copying without understanding. Adopting another protocol's tokenomics without understanding why it works for them and whether it fits your context.

Template for New Protocol

A starting template for protocol tokenomics.

```
tokenomics_template = {  
  'supply': {  
    'total_supply': 1_000_000_000,  
    'initial_circulating_percent': 15,  
    'emission_schedule': 'curve',  
    'emission_duration_years': 4  
  },  
  'distribution': {  
    'community_incentives': 40,
```

```

'team': 15,
'investors': 20,
'treasury': 20,
'initial_liquidity': 5
},
'vesting': {
'team_cliff_months': 12,
'team_vesting_months': 36,
'investor_cliff_months': 6,
'investor_vesting_months': 24
},
'utility': {
'staking_for_rewards': True,
'staking_yield_source': 'protocol_fees',
'fee_discount_for_holders': True,
'governance_voting': True
},
'value_capture': {
'fee_share_to_stakers_percent': 50,
'buyback_burn_percent': 20,
'treasury_percent': 30
},
'mechanisms': {
'minimum_staking_period': '7 days',
'unstaking_period': '7 days',
'vote_escrow': True,
'max_lock_duration': '4 years'
}
}

```

```

def validate_tokenomics(config):

```

```

    issues = []

```

```

    distribution_total = sum(config['distribution'].values())

```

```

    if distribution_total != 100:

```

```

        issues.append(f'Distribution sums to {distribution_total}%,
should be 100%')

```

```

if config['distribution']['team'] > 20:
    issues.append('Team allocation above 20% may concern
investors')

if config['distribution']['community_incentives'] < 30:
    issues.append('Community allocation below 30% may limit
decentralization')

value_capture_total = sum(config['value_capture'].values())
if value_capture_total != 100:
    issues.append(f'Value capture sums to
{value_capture_total}%, should be 100%')

if not config['utility']['staking_for_rewards'] and not
config['utility']['fee_discount_for_holders']:
    issues.append('No clear utility - consider adding staking or
usage benefits')

return {
    'valid': len(issues) == 0,
    'issues': issues
}

```

CHAPTER SUMMARY

Tokenomics determines how value flows through a protocol. Good design aligns incentives among users, holders, and builders. Poor design leaks value or creates unsustainable dynamics.

Supply mechanics include fixed versus unlimited supply, emission schedules, and the critical distinction between total and circulating supply. Understanding dilution from vesting unlocks is essential.

Inflation erodes value when supply grows faster than demand. Deflation through burns or buybacks can increase value but requires sustainable burn sources.

Burning mechanisms include transaction fee burns, protocol revenue burns, and voluntary burns for benefits. Each creates different supply dynamics.

Utility tokens provide functional benefits. Governance tokens provide voting rights. The best designs combine both with fee capture mechanisms.

Staking creates supply sinks and alignment but must offer yields competitive with alternative uses. Liquid staking adds composability at the cost of additional risk.

Modeling tokenomics through simulation helps identify weaknesses before launch. Scenario analysis stress-tests designs under various market conditions.

Practical design requires systematically addressing supply, demand, distribution, value capture, and sustainability while avoiding common mistakes.

The next chapter covers token distribution, examining how to fairly allocate tokens and the mechanisms for launches, airdrops, and vesting.

CHAPTER 7: TOKEN DISTRIBUTION

How tokens reach the hands of users determines a project's trajectory more than almost any other factor. Distribution isn't just logistics -- it's politics, economics, and community building rolled into one critical decision. Get it right and you create loyal stakeholders who defend your protocol. Get it wrong and you breed resentment, dumps, and regulatory scrutiny.

This chapter explores every distribution mechanism in detail. From fair launches that democratize access to complex vesting schedules that align incentives over years. From airdrops that bootstrap communities to liquidity bootstrapping pools that discover prices fairly. You'll understand why projects choose specific mechanisms and how to implement them properly.

SECTION 1: DISTRIBUTION PHILOSOPHY

Before diving into mechanisms, understand why distribution matters so deeply.

The initial distribution creates your stakeholder map. Who holds tokens determines who has power, who captures value, and who cares about your protocol's success. A poorly distributed token concentrates power in few hands, creating dump pressure and centralization concerns. A well-distributed token creates thousands of aligned stakeholders who become evangelists.

Regulatory considerations also matter enormously. The way tokens are distributed affects whether regulators view them as securities. Projects that sell tokens promising future profits look like securities. Projects that distribute tokens to active users who earned them through participation look more like rewards programs.

Distribution Categories

Tokens typically flow to these groups:

Team and founders receive allocation for building the protocol. Usually 15-25% of total supply. These tokens vest over years to ensure long-term commitment. Without meaningful team allocation, founders have no incentive to keep building after launch.

Investors provide capital for development before the protocol generates revenue. Early-stage investors take more risk and typically receive better terms. Seed rounds might get tokens at \$0.01 while public sales price at \$0.10. Investor allocation typically ranges 10-25% of supply.

Community allocation goes to users who interact with the protocol. This includes airdrops, liquidity mining rewards, grants, and ecosystem incentives. The largest portion often goes here -- 40-60% in well-designed tokenomics.

Treasury holds tokens for future needs. Grants, partnerships, emergency reserves, and unforeseen opportunities. Usually controlled by governance or a foundation. Treasury typically holds 10-20% of supply.

Liquidity allocation seeds trading pools so users can buy and sell. Without initial liquidity, tokens are worthless because nobody can trade them. Usually 5-10% of supply.

Distribution Timeline Philosophy

All tokens don't release at once. That would create massive sell pressure as everyone rushes to exit simultaneously. Instead, tokens release gradually through vesting schedules.

The principle: those who need trust should have longer vesting. The team claims to be committed long-term? Prove it with 4-year vesting. Investors claim to be patient capital? Lock them for years too. Only actual users who earned tokens through participation should receive liquid tokens -- they've already proven commitment through action.

This creates an accountability structure. If everyone's tokens are locked, nobody can dump. As tokens gradually unlock, the market can absorb selling pressure without crashing.

SECTION 2: FAIR LAUNCH VS PRESALE

The fundamental choice: do you raise money before launching, or bootstrap entirely from community participation?

Fair Launch Philosophy

Fair launches mean no presale, no investor allocation, no team premine. Everyone starts equal. The most famous fair launch was Bitcoin -- Satoshi mined alongside everyone else with no head start. Dogecoin similarly launched without presale.

Fair launches appeal to crypto's egalitarian ethos. No venture capitalists getting cheap tokens to dump on retail. No insiders with information advantages. Just code released to the world where anyone can participate.

The challenge: how do you fund development without selling tokens? Options include:

Self-funding where founders use personal money. Works for small projects but limits scope.

Grants from foundations or ecosystem funds. Ethereum Foundation, Gitcoin, and similar organizations fund public goods.

Service revenue where you build something useful that generates fees. Use those fees to fund token development.

Retroactive funding where you build first, launch token later, and the token's value rewards past work.

Fair Launch Implementation

A fair launch token typically starts with zero supply and only creates tokens through some action -- mining, staking, or providing liquidity.

```
contract FairLaunchToken {
    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 public totalSupply;

    uint256 public emissionRatePerBlock;
    uint256 public startBlock;
    uint256 public lastMintBlock;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256))
    public allowance;
    mapping(address => uint256) public stakedBalance;
    mapping(address => uint256) public lastClaimBlock;

    address public stakingToken;
    uint256 public totalStaked;

    constructor(
        string memory tokenName,
        string memory tokenSymbol,
        address stakingTokenAddress,
        uint256 emissionRate
    ) {
        name = tokenName;
        symbol = tokenSymbol;
        decimals = 18;
        stakingToken = stakingTokenAddress;
        emissionRatePerBlock = emissionRate;
        startBlock = block.number;
        lastMintBlock = block.number;
    }
}
```

```
function stake(uint256 amount) external {  
    claimRewards();
```

```
    IERC20(stakingToken).transferFrom(msg.sender,  
    address(this), amount);  
    stakedBalance[msg.sender] += amount;  
    totalStaked += amount;  
}
```

```
function unstake(uint256 amount) external {  
    require(stakedBalance[msg.sender] >= amount);
```

```
    claimRewards();
```

```
    stakedBalance[msg.sender] -= amount;  
    totalStaked -= amount;  
    IERC20(stakingToken).transfer(msg.sender, amount);  
}
```

```
function claimRewards() public {  
    uint256 pending = pendingRewards(msg.sender);
```

```
    if (pending > 0) {  
        balanceOf[msg.sender] += pending;  
        totalSupply += pending;  
    }
```

```
    lastClaimBlock[msg.sender] = block.number;  
}
```

```
function pendingRewards(address account) public view  
returns (uint256) {  
    if (stakedBalance[account] == 0 || totalStaked == 0) {  
        return 0;  
    }
```

```
uint256 blocksSinceLastClaim = block.number -  
lastClaimBlock[account];  
uint256 totalEmission = blocksSinceLastClaim *  
emissionRatePerBlock;  
uint256 userShare = (stakedBalance[account] * 1e18) /  
totalStaked;
```

```
return (totalEmission * userShare) / 1e18;  
}
```

```
function transfer(address to, uint256 amount) external  
returns (bool) {  
    require(balanceOf[msg.sender] >= amount);
```

```
    balanceOf[msg.sender] -= amount;  
    balanceOf[to] += amount;
```

```
    return true;  
}
```

```
function approve(address spender, uint256 amount) external  
returns (bool) {  
    allowance[msg.sender][spender] = amount;  
    return true;  
}
```

```
function transferFrom(address from, address to, uint256  
amount) external returns (bool) {  
    require(balanceOf[from] >= amount);  
    require(allowance[from][msg.sender] >= amount);
```

```
    balanceOf[from] -= amount;  
    balanceOf[to] += amount;  
    allowance[from][msg.sender] -= amount;
```

```
    return true;  
}
```

}

This contract creates no tokens at deployment. Tokens only exist when stakers claim rewards. Everyone has equal opportunity to stake and earn.

Presale Models

Most projects need capital to build, so they sell tokens before launch. Presales come in many forms:

Private sale happens first, usually to venture capital firms and angel investors. Prices are lowest because risk is highest. The project might be nothing more than a whitepaper. Private investors often get 50-90% discounts versus public prices.

Seed round follows or overlaps private sale. Slightly higher prices as the project matures. Still significant discounts.

Strategic round targets partners who bring more than money -- exchanges that will list the token, protocols that will integrate, influencers who will promote.

Public sale opens to everyone. Methods include:

ICO (Initial Coin Offering) -- Fixed price sale where anyone can buy. Popular in 2017, now regulatory landmine in many jurisdictions.

IDO (Initial DEX Offering) -- Token launches on decentralized exchange. Often through launchpad platforms.

IEO (Initial Exchange Offering) -- Centralized exchange hosts the sale. Exchange does due diligence and KYC.

SAFT (Simple Agreement for Future Tokens) -- Legal agreement used for US accredited investors. Tokens delivered later when network launches.

Presale Mechanics Implementation

Here's how presales typically work on-chain:

```
contract TokenPresale {  
    address public token;  
    address public paymentToken;  
    address public treasury;  
  
    uint256 public price;  
    uint256 public minPurchase;  
    uint256 public maxPurchase;  
    uint256 public hardCap;  
    uint256 public totalRaised;  
  
    uint256 public startTime;  
    uint256 public endTime;  
  
    bool public finalized;  
    bool public vestingEnabled;  
    uint256 public vestingDuration;  
    uint256 public cliffDuration;  
  
    mapping(address => uint256) public contributions;  
    mapping(address => uint256) public tokenAllocation;  
    mapping(address => uint256) public tokensClaimed;  
    mapping(address => bool) public whitelisted;  
  
    bool public whitelistRequired;  
  
    constructor(  
        address tokenAddress,  
        address paymentTokenAddress,  
        address treasuryAddress,  
        uint256 tokenPrice,  
        uint256 minimum,
```

```
uint256 maximum,  
uint256 cap,  
uint256 start,  
uint256 end  
) {  
    token = tokenAddress;  
    paymentToken = paymentTokenAddress;  
    treasury = treasuryAddress;  
    price = tokenPrice;  
    minPurchase = minimum;  
    maxPurchase = maximum;  
    hardCap = cap;  
    startTime = start;  
    endTime = end;  
}
```

```
function setWhitelistRequired(bool required) external {  
    require(msg.sender == treasury);  
    whitelistRequired = required;  
}
```

```
function addToWhitelist(address[] calldata accounts) external  
{  
    require(msg.sender == treasury);  
  
    for (uint256 i = 0; i < accounts.length; i++) {  
        whitelisted[accounts[i]] = true;  
    }  
}
```

```
function setVesting(uint256 duration, uint256 cliff) external {  
    require(msg.sender == treasury);  
    require(!finalized);  
  
    vestingEnabled = true;  
    vestingDuration = duration;  
    cliffDuration = cliff;
```

```
}
```

```
function contribute(uint256 paymentAmount) external {  
    require(block.timestamp >= startTime);  
    require(block.timestamp <= endTime);  
    require(totalRaised + paymentAmount <= hardCap);  
    require(paymentAmount >= minPurchase);  
    require(contributions[msg.sender] + paymentAmount <=  
maxPurchase);
```

```
    if (whitelistRequired) {  
        require(whitelisted[msg.sender]);  
    }
```

```
    IERC20(paymentToken).transferFrom(msg.sender,  
address(this), paymentAmount);
```

```
    contributions[msg.sender] += paymentAmount;  
    totalRaised += paymentAmount;
```

```
    uint256 tokensToAllocate = (paymentAmount * 1e18) /  
price;  
    tokenAllocation[msg.sender] += tokensToAllocate;  
}
```

```
function finalize() external {  
    require(msg.sender == treasury);  
    require(block.timestamp > endTime || totalRaised >=  
hardCap);  
    require(!finalized);
```

```
    finalized = true;
```

```
    IERC20(paymentToken).transfer(treasury, totalRaised);  
}
```

```
function claimTokens() external {
```

```
require(finalized);
require(tokenAllocation[msg.sender] > 0);

uint256 claimable = getClaimableAmount(msg.sender);
require(claimable > 0);

tokensClaimed[msg.sender] += claimable;
IERC20(token).transfer(msg.sender, claimable);
}
```

```
function getClaimableAmount(address account) public view
returns (uint256) {
    if (!finalized) return 0;
```

```
    uint256 totalAllocation = tokenAllocation[account];
    uint256 alreadyClaimed = tokensClaimed[account];
```

```
    if (!vestingEnabled) {
        return totalAllocation - alreadyClaimed;
    }
```

```
    uint256 vestingStart = endTime;
```

```
    if (block.timestamp < vestingStart + cliffDuration) {
        return 0;
    }
```

```
    uint256 timeVested = block.timestamp - vestingStart;
```

```
    if (timeVested >= vestingDuration) {
        return totalAllocation - alreadyClaimed;
    }
```

```
    uint256 vestedAmount = (totalAllocation * timeVested) /
vestingDuration;
    return vestedAmount - alreadyClaimed;
}
```


}

This contract handles whitelist-controlled sales with optional vesting. Participants contribute payment tokens (usually USDC or ETH) and receive token allocations that vest over time.

SECTION 3: VESTING SCHEDULES

Vesting prevents immediate selling by releasing tokens gradually over time. This aligns incentives -- stakeholders must remain committed to benefit from their full allocation.

Vesting Components

Cliff period -- initial lockup where nothing vests. A 1-year cliff means zero tokens for 12 months. After the cliff, accumulated vesting releases.

Vesting duration -- total time until fully vested. 4-year vesting with 1-year cliff means nothing for year 1, then linear release over remaining 3 years.

Vesting schedule -- how tokens release. Linear vesting releases constant amounts. Monthly vesting releases chunks each month. Quarterly, annually, or custom schedules also exist.

Initial unlock -- some tokens release immediately at TGE (Token Generation Event). Common for investors who negotiate partial liquidity.

Standard Vesting Patterns

Team tokens typically: 4-year total vesting, 1-year cliff, then monthly or continuous release. This mirrors equity vesting in startups.

Investor tokens vary by round:

- Seed: 2-4 year vesting, 6-12 month cliff, 5-20% TGE unlock
- Private: 1-2 year vesting, 3-6 month cliff, 10-25% TGE unlock
- Public: 0-12 month vesting, sometimes no cliff, 20-100% TGE unlock

Community tokens: Usually shorter vesting or immediate distribution. Users earned these through action, so long locks feel punitive.

Vesting Contract Implementation

```
contract TokenVesting {
    struct VestingSchedule {
        uint256 totalAmount;
        uint256 startTime;
        uint256 cliffDuration;
        uint256 vestingDuration;
        uint256 initialUnlock;
        uint256 amountClaimed;
        bool revocable;
        bool revoked;
    }

    address public token;
    address public admin;

    mapping(address => VestingSchedule[]) public
    vestingSchedules;

    constructor(address tokenAddress) {
        token = tokenAddress;
        admin = msg.sender;
    }

    function createVestingSchedule(
```

```

address beneficiary,
uint256 totalAmount,
uint256 startTime,
uint256 cliffDuration,
uint256 vestingDuration,
uint256 initialUnlockPercent,
bool revocable
) external {
    require(msg.sender == admin);
    require(vestingDuration > 0);
    require(initialUnlockPercent <= 100);

    uint256 initialAmount = (totalAmount *
initialUnlockPercent) / 100;

    IERC20(token).transferFrom(msg.sender, address(this),
totalAmount);

    VestingSchedule memory schedule = VestingSchedule({
totalAmount: totalAmount,
startTime: startTime,
cliffDuration: cliffDuration,
vestingDuration: vestingDuration,
initialUnlock: initialAmount,
amountClaimed: 0,
revocable: revocable,
revoked: false
});

    vestingSchedules[beneficiary].push(schedule);
}

function claim(uint256 scheduleIndex) external {
    VestingSchedule storage schedule =
vestingSchedules[msg.sender][scheduleIndex];

    require(!schedule.revoked);

```

```
uint256 claimable = getClaimableAmount(msg.sender,  
scheduleIndex);  
require(claimable > 0);
```

```
schedule.amountClaimed += claimable;  
IERC20(token).transfer(msg.sender, claimable);  
}
```

```
function claimAll() external {  
uint256 totalClaimable = 0;
```

```
for (uint256 i = 0; i < vestingSchedules[msg.sender].length;  
i++) {
```

```
VestingSchedule storage schedule =  
vestingSchedules[msg.sender][i];
```

```
if (schedule.revoked) continue;
```

```
uint256 claimable = getClaimableAmount(msg.sender, i);  
schedule.amountClaimed += claimable;  
totalClaimable += claimable;  
}
```

```
require(totalClaimable > 0);  
IERC20(token).transfer(msg.sender, totalClaimable);  
}
```

```
function getClaimableAmount(  
address beneficiary,  
uint256 scheduleIndex  
) public view returns (uint256) {  
VestingSchedule memory schedule =  
vestingSchedules[beneficiary][scheduleIndex];
```

```
if (schedule.revoked) return 0;
```

```
uint256 vested = getVestedAmount(beneficiary,  
scheduleIndex);  
return vested - schedule.amountClaimed;  
}
```

```
function getVestedAmount(  
address beneficiary,  
uint256 scheduleIndex  
) public view returns (uint256) {  
VestingSchedule memory schedule =  
vestingSchedules[beneficiary][scheduleIndex];
```

```
if (block.timestamp < schedule.startTime) {  
return 0;  
}
```

```
if (block.timestamp < schedule.startTime +  
schedule.cliffDuration) {  
return schedule.initialUnlock;  
}
```

```
uint256 timeFromStart = block.timestamp -  
schedule.startTime;
```

```
if (timeFromStart >= schedule.vestingDuration) {  
return schedule.totalAmount;  
}
```

```
uint256 vestingAmount = schedule.totalAmount -  
schedule.initialUnlock;  
uint256 vestedPortion = (vestingAmount * timeFromStart) /  
schedule.vestingDuration;
```

```
return schedule.initialUnlock + vestedPortion;  
}
```

```
function revoke(address beneficiary, uint256 scheduleIndex)
```

```

external {
    require(msg.sender == admin);

    VestingSchedule storage schedule =
vestingSchedules[beneficiary][scheduleIndex];
    require(schedule.revocable);
    require(!schedule.revoked);

    uint256 vested = getVestedAmount(beneficiary,
scheduleIndex);
    uint256 refund = schedule.totalAmount - vested;

    schedule.revoked = true;

    if (vested > schedule.amountClaimed) {
        IERC20(token).transfer(beneficiary, vested -
schedule.amountClaimed);
    }

    if (refund > 0) {
        IERC20(token).transfer(admin, refund);
    }
}

function getScheduleCount(address beneficiary) external view
returns (uint256) {
    return vestingSchedules[beneficiary].length;
}
}

```

This contract supports multiple vesting schedules per address (useful when someone participates in multiple rounds), initial unlocks, cliffs, and revocable grants for employees.

Vesting Dashboard Integration

Users need to see their vesting status. Here's how to build a

dashboard that displays vesting information:

```
import { ethers } from 'ethers'
```

```
const VESTING_ABI = [  
  'function getScheduleCount(address) view returns (uint256)',  
  'function vestingSchedules(address, uint256) view returns  
(uint256 totalAmount, uint256 startTime, uint256  
cliffDuration, uint256 vestingDuration, uint256 initialUnlock,  
uint256 amountClaimed, bool revocable, bool revoked)',  
  'function getVestedAmount(address, uint256) view returns  
(uint256)',  
  'function getClaimableAmount(address, uint256) view returns  
(uint256)',  
  'function claim(uint256)',  
  'function claimAll()' ]
```

```
class VestingDashboard {  
  constructor(vestingAddress, provider) {  
    this.contract = new ethers.Contract(vestingAddress,  
VESTING_ABI, provider)  
  }
```

```
  async getVestingSchedules(userAddress) {  
    const count = await  
this.contract.getScheduleCount(userAddress)  
    const schedules = []
```

```
    for (let i = 0; i < count; i++) {  
      const schedule = await  
this.contract.vestingSchedules(userAddress, i)  
      const vested = await  
this.contract.getVestedAmount(userAddress, i)  
      const claimable = await  
this.contract.getClaimableAmount(userAddress, i)
```

```

schedules.push({
  index: i,
  totalAmount: ethers.formatEther(schedule.totalAmount),
  startTime: new Date(Number(schedule.startTime) * 1000),
  cliffEnd: new Date((Number(schedule.startTime) +
    Number(schedule.cliffDuration)) * 1000),
  vestingEnd: new Date((Number(schedule.startTime) +
    Number(schedule.vestingDuration)) * 1000),
  initialUnlock: ethers.formatEther(schedule.initialUnlock),
  claimed: ethers.formatEther(schedule.amountClaimed),
  vested: ethers.formatEther(vested),
  claimable: ethers.formatEther(claimable),
  revocable: schedule.revocable,
  revoked: schedule.revoked,
  percentVested: (Number(vested) /
    Number(schedule.totalAmount) * 100).toFixed(2)
})
}

```

```

return schedules
}

```

```

async getTotalClaimable(userAddress) {
  const schedules = await
  this.getVestingSchedules(userAddress)
  return schedules.reduce((sum, s) => sum +
    parseFloat(s.claimable), 0)
}

```

```

async claimAll(signer) {
  const contractWithSigner = this.contract.connect(signer)
  const tx = await contractWithSigner.claimAll()
  return tx.wait()
}

```

```

getVestingTimeline(schedule) {
  const now = Date.now()

```



```
const startMs = schedule.startTime.getTime()
const cliffMs = schedule.cliffEnd.getTime()
const endMs = schedule.vestingEnd.getTime()
```

```
if (now < startMs) {
  return { status: 'not_started', message: 'Vesting has not started'
}
}
```

```
if (now < cliffMs) {
  const daysUntilCliff = Math.ceil((cliffMs - now) / (1000 * 60
* 60 * 24))
  return { status: 'cliff', message: `${daysUntilCliff} days until
cliff ends` }
}
```

```
if (now < endMs) {
  const daysUntilEnd = Math.ceil((endMs - now) / (1000 * 60
* 60 * 24))
  return { status: 'vesting', message: `${daysUntilEnd} days
until fully vested` }
}
```

```
return { status: 'complete', message: 'Fully vested' }
}
}
```

```
async function displayVestingInfo(dashboard, userAddress) {
  const schedules = await
dashboard.getVestingSchedules(userAddress)
```

```
  console.log('Vesting Schedules:')
  console.log('=' .repeat(50))
```

```
  for (const schedule of schedules) {
    console.log(` Schedule ${schedule.index}:`)
    console.log(`   Total: ${schedule.totalAmount} tokens`)
  }
```

```
console.log(` Vested: ${schedule.vested}
(${schedule.percentVested}%)`)
console.log(` Claimed: ${schedule.claimed}`)
console.log(` Claimable: ${schedule.claimable}`)

const timeline = dashboard.getVestingTimeline(schedule)
console.log(` Status: ${timeline.message}`)
console.log("")
}

const totalClaimable = await
dashboard.getTotalClaimable(userAddress)
console.log(` Total Claimable: ${totalClaimable} tokens`)
}
```

SECTION 4: AIRDROP MECHANICS

Airdrops distribute tokens to existing wallet addresses, usually based on past activity. The term comes from tokens seemingly dropping from the sky into wallets.

Airdrop Strategies

Retroactive airdrops reward past users. Uniswap famously airdropped UNI to everyone who had used the protocol. This rewards early adopters and creates instant stakeholders.

Snapshot airdrops capture wallet states at a specific block. Hold Token A at block X, receive Token B proportionally. Simple but gameable once announced.

Activity-based airdrops reward specific actions. Used the protocol more than 10 times? More than 5 different features? Been active for more than 6 months? Each criterion adds multipliers.

Tiered airdrops give different amounts based on user category. Power users get more. Long-term users get more. Contributors to ecosystem get bonuses.

Sybil Resistance

The biggest challenge with airdrops: preventing one person from claiming with thousands of wallets. If I know an airdrop is coming, I create 1000 wallets and do minimal activity with each.

Sybil resistance strategies:

On-chain clustering -- analyze transaction patterns to identify wallets controlled by same entity. Shared funding sources, similar transaction timing, funds flowing between addresses.

Minimum activity thresholds -- require meaningful activity impossible to fake cheaply. Minimum transaction count, minimum volume, minimum time active.

Identity verification -- require proof of humanity through services like Bitcoin Passport, BrightID, or Worldcoin. One human, one allocation.

Social credentials -- require ENS name, Twitter account, Discord participation, GitHub contributions. Harder to fake at scale.

Gas cost requirements -- require actions that cost gas. Creating 1000 wallets that each spend \$50 in gas costs \$50,000.

Airdrop Contract Implementation

```
contract MerkleAirdrop {  
    address public token;  
    bytes32 public merkleRoot;
```

```
uint256 public expiryTime;
```

```
mapping(address => bool) public hasClaimed;
```

```
constructor(  
    address tokenAddress,  
    bytes32 root,  
    uint256 expiry  
) {  
    token = tokenAddress;  
    merkleRoot = root;  
    expiryTime = expiry;  
}
```

```
function claim(  
    uint256 amount,  
    bytes32[] calldata merkleProof  
) external {  
    require(!hasClaimed[msg.sender]);  
    require(block.timestamp < expiryTime);
```

```
    bytes32 leaf = keccak256(abi.encodePacked(msg.sender,  
    amount));  
    require(verifyProof(merkleProof, merkleRoot, leaf));
```

```
    hasClaimed[msg.sender] = true;  
    IERC20(token).transfer(msg.sender, amount);  
}
```

```
function verifyProof(  
    bytes32[] memory proof,  
    bytes32 root,  
    bytes32 leaf  
) internal pure returns (bool) {  
    bytes32 computedHash = leaf;
```

```
    for (uint256 i = 0; i < proof.length; i++) {
```

```

bytes32 proofElement = proof[i];

if (computedHash <= proofElement) {
  computedHash =
keccak256(abi.encodePacked(computedHash, proofElement));
} else {
  computedHash =
keccak256(abi.encodePacked(proofElement, computedHash));
}
}

return computedHash == root;
}

function withdrawExpired() external {
  require(block.timestamp >= expiryTime);

  uint256 remaining =
IERC20(token).balanceOf(address(this));
  IERC20(token).transfer(msg.sender, remaining);
}
}

```

Merkle trees enable gas-efficient airdrops. Instead of storing every eligible address on-chain (expensive), store only the merkle root. Users provide proofs to claim.

Generating Merkle Trees

Here's how to generate the merkle tree and proofs off-chain:

```

import { StandardMerkleTree } from '@openzeppelin/merkle-tree'
import fs from 'fs'

function generateAirdropMerkleTree(recipients) {
  const values = recipients.map(r => [r.address,

```



```
score += min(features_used * 15, 75)
```

```
first_interaction = self.get_first_interaction(address)
if first_interaction and first_interaction < start_block +
1000000:
```

```
score += 50
```

```
return score
```

```
def get_protocol_transactions(self, address, start_block,
end_block):
    count = 0
    for contract_address in self.contracts:
        count += self.count_interactions(address, contract_address,
start_block, end_block)
    return count
```

```
def count_interactions(self, user, contract, start_block,
end_block):
    return 0
```

```
def get_active_days(self, address, start_block, end_block):
    return 0
```

```
def get_total_volume(self, address, start_block, end_block):
    return 0
```

```
def get_features_used(self, address, start_block, end_block):
    return 0
```

```
def get_first_interaction(self, address):
    return None
```

```
def calculate_allocation(self, score, total_tokens, max_score):
if score < 10:
    return 0
```



```
if score >= 200:
    tier = 'gold'
    multiplier = 4
elif score >= 100:
    tier = 'silver'
    multiplier = 2
elif score >= 50:
    tier = 'bronze'
    multiplier = 1
else:
    tier = 'base'
    multiplier = 0.5
```

```
base_allocation = total_tokens / 100000
return int(base_allocation * multiplier)
```

```
def generate_airdrop_list(self, addresses, snapshot_block,
total_tokens):
    results = []
```

```
    for address in addresses:
        score = self.calculate_activity_score(address, 0,
snapshot_block)
        allocation = self.calculate_allocation(score, total_tokens, 500)
```

```
    if allocation > 0:
        results.append({
            'address': address,
            'score': score,
            'allocation': allocation
        })
```

```
    return sorted(results, key=lambda x: x['allocation'],
reverse=True)
```

SECTION 5: LIQUIDITY BOOTSTRAPPING

New tokens face a chicken-and-egg problem: nobody wants to buy a token with no liquidity, but you can't have liquidity without buyers. Liquidity bootstrapping solves this through specialized mechanisms.

Liquidity Bootstrapping Pools (LBP)

LBPs use dynamic weights that shift over time. Initially, the token weight might be 90% while the paired asset (ETH/USDC) is 10%. Weights gradually shift to 50/50 or similar.

This creates automatic price discovery and discourages bots:

- Early buyers pay higher prices but get more tokens per dollar due to low liquidity token weight
- Price naturally decreases as weights shift unless buying pressure maintains it
- Bots can't frontrun effectively because price changes continuously
- Snipers who buy at launch face immediate price drop as weights shift

```
contract LiquidityBootstrappingPool {  
    address public projectToken;  
    address public baseToken;
```

```
    uint256 public projectTokenBalance;  
    uint256 public baseTokenBalance;
```

```
    uint256 public startWeight;  
    uint256 public endWeight;  
    uint256 public startTime;  
    uint256 public endTime;
```

```
    uint256 constant WEIGHT_PRECISION = 1e18;
```

```
    bool public finalized;
```

address public owner;

```
constructor(  
  address project,  
  address base,  
  uint256 initialProjectWeight,  
  uint256 finalProjectWeight,  
  uint256 start,  
  uint256 end  
) {  
  projectToken = project;  
  baseToken = base;  
  startWeight = initialProjectWeight;  
  endWeight = finalProjectWeight;  
  startTime = start;  
  endTime = end;  
  owner = msg.sender;  
}
```

```
function getCurrentWeight() public view returns (uint256  
projectWeight, uint256 baseWeight) {  
  if (block.timestamp <= startTime) {  
    return (startWeight, WEIGHT_PRECISION - startWeight);  
  }
```

```
  if (block.timestamp >= endTime) {  
    return (endWeight, WEIGHT_PRECISION - endWeight);  
  }
```

```
  uint256 elapsed = block.timestamp - startTime;  
  uint256 duration = endTime - startTime;
```

```
  uint256 weightChange;  
  if (startWeight > endWeight) {  
    weightChange = ((startWeight - endWeight) * elapsed) /  
duration;  
    projectWeight = startWeight - weightChange;
```

```
} else {  
    weightChange = ((endWeight - startWeight) * elapsed) /  
    duration;  
    projectWeight = startWeight + weightChange;  
}
```

```
baseWeight = WEIGHT_PRECISION - projectWeight;  
return (projectWeight, baseWeight);  
}
```

```
function getSpotPrice() public view returns (uint256) {  
    (uint256 projectWeight, uint256 baseWeight) =  
    getCurrentWeight();
```

```
    uint256 numerator = (baseTokenBalance *  
    WEIGHT_PRECISION) / baseWeight;  
    uint256 denominator = (projectTokenBalance *  
    WEIGHT_PRECISION) / projectWeight;
```

```
    return (numerator * WEIGHT_PRECISION) / denominator;  
}
```

```
function calculateSwap(  
    uint256 baseTokensIn  
    ) public view returns (uint256 projectTokensOut) {  
    (uint256 projectWeight, uint256 baseWeight) =  
    getCurrentWeight();
```

```
    uint256 newBaseBalance = baseTokenBalance +  
    baseTokensIn;  
    uint256 baseRatio = (newBaseBalance *  
    WEIGHT_PRECISION) / baseTokenBalance;
```

```
    uint256 exponent = (baseWeight * WEIGHT_PRECISION) /  
    projectWeight;  
    uint256 priceRatio = power(baseRatio, exponent);
```

```
uint256 newProjectBalance = (projectTokenBalance *  
WEIGHT_PRECISION) / priceRatio;  
projectTokensOut = projectTokenBalance -  
newProjectBalance;
```

```
return projectTokensOut;  
}
```

```
function power(uint256 base, uint256 exp) internal pure  
returns (uint256) {  
    return base;  
}
```

```
function swap(uint256 baseTokensIn, uint256  
minProjectTokensOut) external {  
    require(block.timestamp >= startTime);  
    require(block.timestamp <= endTime);  
    require(!finalized);
```

```
    uint256 projectTokensOut = calculateSwap(baseTokensIn);  
    require(projectTokensOut >= minProjectTokensOut);
```

```
    IERC20(baseToken).transferFrom(msg.sender, address(this),  
baseTokensIn);  
    baseTokenBalance += baseTokensIn;
```

```
    projectTokenBalance -= projectTokensOut;  
    IERC20(projectToken).transfer(msg.sender,  
projectTokensOut);  
}
```

```
function seedLiquidity(uint256 projectAmount, uint256  
baseAmount) external {  
    require(msg.sender == owner);  
    require(projectTokenBalance == 0);
```

```
    IERC20(projectToken).transferFrom(msg.sender,
```

```

address(this), projectAmount);
IERC20(baseToken).transferFrom(msg.sender, address(this),
baseAmount);

projectTokenBalance = projectAmount;
baseTokenBalance = baseAmount;
}

function finalize() external {
require(msg.sender == owner);
require(block.timestamp > endTime);

finalized = true;

IERC20(baseToken).transfer(owner, baseTokenBalance);

if (projectTokenBalance > 0) {
IERC20(projectToken).transfer(owner, projectTokenBalance);
}
}
}

```

Initial DEX Offering (IDO) Platforms

IDO platforms host token launches for projects. They provide:

- Existing user base of potential buyers
- Technical infrastructure for sales
- Due diligence (varying quality)
- Lottery or staking systems for allocation

Popular IDO platforms include DAO Maker, Seedify, GameFi, and chain-specific options like Raydium on Solana.

Participating in IDOs programmatically:

```

import { ethers } from 'ethers'

const IDO_PLATFORM_ABI = [

```

```

'function getPoolInfo(uint256 poolId) view returns (address
token, uint256 price, uint256 totalRaise, uint256 hardCap,
uint256 startTime, uint256 endTime)',
'function getUserAllocation(uint256 poolId, address user)
view returns (uint256)',
'function participate(uint256 poolId, uint256 amount)',
'function claim(uint256 poolId)',
'function userParticipation(uint256 poolId, address user) view
returns (uint256 amount, bool claimed)'
]

```

```

class IDOParticipant {
  constructor(platformAddress, provider) {
    this.contract = new ethers.Contract(platformAddress,
IDO_PLATFORM_ABI, provider)
  }

```

```

  async getPoolDetails(poolId) {
    const info = await this.contract.getPoolInfo(poolId)
    return {
      token: info.token,
      price: ethers.formatEther(info.price),
      totalRaise: ethers.formatEther(info.totalRaise),
      hardCap: ethers.formatEther(info.hardCap),
      startTime: new Date(Number(info.startTime) * 1000),
      endTime: new Date(Number(info.endTime) * 1000)
    }
  }

```

```

  async checkAllocation(poolId, userAddress) {
    const allocation = await
this.contract.getUserAllocation(poolId, userAddress)
    return ethers.formatEther(allocation)
  }

```

```

  async participate(poolId, amount, signer) {
    const contractWithSigner = this.contract.connect(signer)

```

```

const tx = await contractWithSigner.participate(poolId,
ethers.parseEther(amount.toString()))
return tx.wait()
}

async claimTokens(poolId, signer) {
const contractWithSigner = this.contract.connect(signer)
const tx = await contractWithSigner.claim(poolId)
return tx.wait()
}
}

```

SECTION 6: TOKEN GENERATION EVENTS

The Token Generation Event (TGE) is the moment tokens become tradable. Everything before TGE is preparation. Everything after is market dynamics.

TGE Checklist

Before TGE:

Smart contracts deployed and verified. All token contracts, vesting contracts, staking contracts ready and audited.

Liquidity prepared. DEX pools seeded or LBP configured. Centralized exchange listings confirmed.

Distribution contracts funded. Airdrop contracts have tokens. Vesting contracts have allocations.

Security review complete. Audits published. Bug bounty active. Multisig controls verified.

Legal review complete. Token classification understood. Jurisdictional restrictions implemented.

Marketing prepared. Announcements scheduled. Community briefed. Documentation published.

Technical monitoring ready. Price feeds, analytics, alert systems active.

TGE Execution Contract

```
contract TokenGenerationEvent {  
    address public token;  
    address public admin;
```

```
    bool public tgeExecuted;  
    uint256 public tgeTimestamp;
```

```
    address public vestingContract;  
    address public airdropContract;  
    address public liquidityContract;  
    address public treasuryWallet;
```

```
    struct Allocation {  
        address recipient;  
        uint256 amount;  
        string category;  
    }
```

```
    Allocation[] public allocations;  
    mapping(string => uint256) public categoryTotals;
```

```
    constructor(address tokenAddress) {  
        token = tokenAddress;  
        admin = msg.sender;  
    }
```

```
    function setContracts(  
        address vesting,  
        address airdrop,
```

```
address liquidity,  
address treasury  
) external {  
    require(msg.sender == admin);  
    require(!tgeExecuted);
```

```
vestingContract = vesting;  
airdropContract = airdrop;  
liquidityContract = liquidity;  
treasuryWallet = treasury;  
}
```

```
function addAllocation(  
    address recipient,  
    uint256 amount,  
    string calldata category  
) external {  
    require(msg.sender == admin);  
    require(!tgeExecuted);
```

```
    allocations.push(Allocation({  
        recipient: recipient,  
        amount: amount,  
        category: category  
    }));
```

```
    categoryTotals[category] += amount;  
}
```

```
function executeTGE() external {  
    require(msg.sender == admin);  
    require(!tgeExecuted);  
    require(vestingContract != address(0));  
    require(airdropContract != address(0));  
    require(liquidityContract != address(0));
```

```
    tgeExecuted = true;
```

```

tgeTimestamp = block.timestamp;

for (uint256 i = 0; i < allocations.length; i++) {
    Allocation memory alloc = allocations[i];
    IERC20(token).transfer(alloc.recipient, alloc.amount);
}
}

function getAllocationCount() external view returns (uint256)
{
    return allocations.length;
}

function getCategoryTotal(string calldata category) external
view returns (uint256) {
    return categoryTotals[category];
}
}

```

TGE Monitoring

Track everything at TGE:

```

import { ethers } from 'ethers'

class TGEMonitor {
    constructor(provider, tokenAddress, dexPairAddress) {
        this.provider = provider
        this.tokenAddress = tokenAddress
        this.dexPairAddress = dexPairAddress

        this.tokenContract = new ethers.Contract(
            tokenAddress,
            ['event Transfer(address indexed from, address indexed to,
            uint256 value)'],
            provider
        )
    }
}

```

```
}
```

```
async monitorTransfers(callback) {  
  this.tokenContract.on('Transfer', (from, to, value, event) =>  
  {  
    callback({  
      type: 'transfer',  
      from,  
      to,  
      amount: ethers.formatEther(value),  
      txHash: event.log.transactionHash,  
      block: event.log.blockNumber  
    })  
  })  
}
```

```
async getHolderDistribution() {  
  const filter = this.tokenContract.filters.Transfer()  
  const events = await this.tokenContract.queryFilter(filter)
```

```
  const balances = {}
```

```
  for (const event of events) {  
    const from = event.args.from  
    const to = event.args.to  
    const amount = event.args.value
```

```
    if (from !== ethers.ZeroAddress) {  
      balances[from] = (balances[from] || 0n) - amount  
    }  
    balances[to] = (balances[to] || 0n) + amount  
  }
```

```
  const holders = Object.entries(balances)  
    .filter(([addr, bal]) => bal > 0n)  
    .sort((a, b) => Number(b[1] - a[1]))
```

```

return holders.map(([address, balance]) => ({
  address,
  balance: ethers.formatEther(balance)
}))
}

```

```

async checkConcentration() {
  const holders = await this.getHolderDistribution()
  const totalSupply = holders.reduce((sum, h) => sum +
parseFloat(h.balance), 0)

```

```

  const top10 = holders.slice(0, 10)
  const top10Balance = top10.reduce((sum, h) => sum +
parseFloat(h.balance), 0)
  const top10Percent = (top10Balance / totalSupply *
100).toFixed(2)

```

```

  const top100 = holders.slice(0, 100)
  const top100Balance = top100.reduce((sum, h) => sum +
parseFloat(h.balance), 0)
  const top100Percent = (top100Balance / totalSupply *
100).toFixed(2)

```

```

  return {
    totalHolders: holders.length,
    top10Concentration: top10Percent,
    top100Concentration: top100Percent,
    largestHolder: holders[0]
  }
}
}

```

SECTION 7: POST-TGE DISTRIBUTION MANAGEMENT

Distribution doesn't end at TGE. Ongoing programs continue distributing tokens:

Liquidity Mining Programs

Reward users who provide liquidity to trading pools:

```
contract LiquidityMiningProgram {
    address public rewardToken;
    address public lpToken;

    uint256 public rewardPerSecond;
    uint256 public startTime;
    uint256 public endTime;

    uint256 public accRewardPerShare;
    uint256 public lastRewardTime;
    uint256 public totalStaked;

    uint256 constant PRECISION = 1e18;

    struct UserInfo {
        uint256 amount;
        uint256 rewardDebt;
    }

    mapping(address => UserInfo) public userInfo;

    constructor(
        address reward,
        address lp,
        uint256 rewardRate,
        uint256 start,
        uint256 end
    ) {
        rewardToken = reward;
        lpToken = lp;
        rewardPerSecond = rewardRate;
        startTime = start;
```

```
endTime = end;  
lastRewardTime = start;  
}
```

```
function updatePool() public {  
    if (block.timestamp <= lastRewardTime) {  
        return;  
    }
```

```
    if (totalStaked == 0) {  
        lastRewardTime = block.timestamp;  
        return;  
    }
```

```
    uint256 endTs = block.timestamp > endTime ? endTime :  
        block.timestamp;  
    uint256 startTs = lastRewardTime < startTime ? startTime :  
        lastRewardTime;
```

```
    if (endTs <= startTs) {  
        return;  
    }
```

```
    uint256 duration = endTs - startTs;  
    uint256 reward = duration * rewardPerSecond;
```

```
    accRewardPerShare += (reward * PRECISION) /  
        totalStaked;  
    lastRewardTime = block.timestamp;  
}
```

```
function deposit(uint256 amount) external {  
    updatePool();
```

```
    UserInfo storage user = userInfo[msg.sender];
```

```
    if (user.amount > 0) {
```

```
uint256 pending = (user.amount * accRewardPerShare) /  
PRECISION - user.rewardDebt;  
if (pending > 0) {  
    IERC20(rewardToken).transfer(msg.sender, pending);  
}  
}
```

```
if (amount > 0) {  
    IERC20(lpToken).transferFrom(msg.sender, address(this),  
amount);  
    user.amount += amount;  
    totalStaked += amount;  
}
```

```
user.rewardDebt = (user.amount * accRewardPerShare) /  
PRECISION;  
}
```

```
function withdraw(uint256 amount) external {  
    UserInfo storage user = userInfo[msg.sender];  
    require(user.amount >= amount);
```

```
    updatePool();
```

```
    uint256 pending = (user.amount * accRewardPerShare) /  
PRECISION - user.rewardDebt;
```

```
    if (pending > 0) {  
        IERC20(rewardToken).transfer(msg.sender, pending);  
    }
```

```
    if (amount > 0) {  
        user.amount -= amount;  
        totalStaked -= amount;  
        IERC20(lpToken).transfer(msg.sender, amount);  
    }
```



```

    user.rewardDebt = (user.amount * accRewardPerShare) /
PRECISION;
}

function pendingReward(address account) external view
returns (uint256) {
    UserInfo memory user = userInfo[account];

    uint256 currentAccReward = accRewardPerShare;

    if (block.timestamp > lastRewardTime && totalStaked > 0)
    {
        uint256 endTs = block.timestamp > endTime ? endTime :
block.timestamp;
        uint256 startTs = lastRewardTime < startTime ? startTime :
lastRewardTime;

        if (endTs > startTs) {
            uint256 duration = endTs - startTs;
            uint256 reward = duration * rewardPerSecond;
            currentAccReward += (reward * PRECISION) / totalStaked;
        }
    }

    return (user.amount * currentAccReward) / PRECISION -
user.rewardDebt;
}
}

```

Grants and Ecosystem Funding

Treasuries fund development through grants:

```

contract GrantsProgram {
    address public token;
    address public committee;
}

```

```
uint256 public totalBudget;  
uint256 public spent;
```

```
struct Grant {  
    address recipient;  
    uint256 amount;  
    uint256 amountPaid;  
    string description;  
    uint256 createdAt;  
    bool approved;  
    bool cancelled;  
    uint256[] milestones;  
    uint256 milestonesCompleted;  
}
```

```
Grant[] public grants;
```

```
mapping(address => uint256[]) public recipientGrants;
```

```
constructor(address tokenAddress, uint256 budget) {  
    token = tokenAddress;  
    committee = msg.sender;  
    totalBudget = budget;  
}
```

```
function proposeGrant(  
    address recipient,  
    uint256 amount,  
    string calldata description,  
    uint256[] calldata milestones  
) external {  
    require(msg.sender == committee);
```

```
    uint256 totalMilestones = 0;  
    for (uint256 i = 0; i < milestones.length; i++) {  
        totalMilestones += milestones[i];  
    }
```

```
require(totalMilestones == amount);
```

```
uint256 grantId = grants.length;
```

```
grants.push(Grant({  
  recipient: recipient,  
  amount: amount,  
  amountPaid: 0,  
  description: description,  
  createdAt: block.timestamp,  
  approved: false,  
  cancelled: false,  
  milestones: milestones,  
  milestonesCompleted: 0  
}));
```

```
recipientGrants[recipient].push(grantId);  
}
```

```
function approveGrant(uint256 grantId) external {  
  require(msg.sender == committee);
```

```
  Grant storage grant = grants[grantId];  
  require(!grant.approved);  
  require(!grant.cancelled);  
  require(spent + grant.amount <= totalBudget);
```

```
  grant.approved = true;  
  spent += grant.amount;  
}
```

```
function completeMilestone(uint256 grantId) external {  
  require(msg.sender == committee);
```

```
  Grant storage grant = grants[grantId];  
  require(grant.approved);  
  require(!grant.cancelled);
```

```

require(grant.milestonesCompleted <
grant.milestones.length);

uint256 milestoneIndex = grant.milestonesCompleted;
uint256 payment = grant.milestones[milestoneIndex];

grant.milestonesCompleted + +;
grant.amountPaid += payment;

IERC20(token).transfer(grant.recipient, payment);
}

function cancelGrant(uint256 grantId) external {
require(msg.sender == committee);

Grant storage grant = grants[grantId];
require(!grant.cancelled);

grant.cancelled = true;

uint256 unspent = grant.amount - grant.amountPaid;
spent -= unspent;
}

function getGrantCount() external view returns (uint256) {
return grants.length;
}

function getRecipientGrants(address recipient) external view
returns (uint256[] memory) {
return recipientGrants[recipient];
}
}

```

SECTION 8: DISTRIBUTION ANALYTICS

Track distribution health over time:

```

class DistributionAnalytics:
    def __init__(self, token_data):
        self.holders = token_data['holders']
        self.total_supply = token_data['total_supply']
        self.vesting_schedules = token_data['vesting']
        self.emissions = token_data['emissions']

    def calculate_gini_coefficient(self):
        balances = sorted([h['balance'] for h in self.holders])
        n = len(balances)

        if n == 0:
            return 0

        cumulative = 0
        for i, balance in enumerate(balances):
            cumulative += (2 * (i + 1) - n - 1) * balance

        total = sum(balances)
        if total == 0:
            return 0

        gini = cumulative / (n * total)
        return gini

    def get_distribution_metrics(self):
        balances = sorted([h['balance'] for h in self.holders],
                           reverse=True)
        total = sum(balances)

        if total == 0:
            return {}

        cumulative = 0
        top_1_percent_index = max(1, len(balances) // 100)
        top_10_percent_index = max(1, len(balances) // 10)

```

```
top_1_balance = sum(balances[:top_1_percent_index])
top_10_balance = sum(balances[:top_10_percent_index])
```

```
return {
    'total_holders': len(balances),
    'gini_coefficient': self.calculate_gini_coefficient(),
    'top_1_percent_holdings': top_1_balance / total * 100,
    'top_10_percent_holdings': top_10_balance / total * 100,
    'median_balance': balances[len(balances) // 2] if balances
else 0,
    'average_balance': total / len(balances) if balances else 0
}
```

```
def project_unlock_schedule(self, months_ahead):
    schedule = []
    current_circulating = self.get_current_circulating()
```

```
    for month in range(months_ahead):
        monthly_unlock = self.calculate_monthly_unlock(month)
        monthly_emission = self.calculate_monthly_emission(month)
```

```
    total_new_supply = monthly_unlock + monthly_emission
    current_circulating += total_new_supply
```

```
    schedule.append({
        'month': month + 1,
        'new_unlocks': monthly_unlock,
        'new_emissions': monthly_emission,
        'total_circulating': current_circulating,
        'percent_of_total': current_circulating / self.total_supply * 100
    })
```

```
    return schedule
```

```
def get_current_circulating(self):
    locked = sum(v['remaining'] for v in self.vesting_schedules)
```

```
return self.total_supply - locked
```

```
def calculate_monthly_unlock(self, month_offset):  
    return 0
```

```
def calculate_monthly_emission(self, month_offset):  
    return 0
```

```
def identify_concentration_risks(self):  
    risks = []
```

```
    metrics = self.get_distribution_metrics()
```

```
    if metrics.get('gini_coefficient', 0) > 0.9:  
        risks.append({  
            'type': 'high_concentration',  
            'severity': 'high',  
            'description': 'Token distribution is highly concentrated'  
        })
```

```
    top_holder_balance = self.holders[0]['balance'] if self.holders  
    else 0
```

```
    if top_holder_balance / self.total_supply > 0.1:  
        risks.append({  
            'type': 'whale_risk',  
            'severity': 'medium',  
            'description': f'Single holder owns more than 10% of supply'  
        })
```

```
    schedule = self.project_unlock_schedule(3)
```

```
    for month_data in schedule:
```

```
        if month_data['new_unlocks'] / month_data['total_circulating']  
        > 0.1:
```

```
            risks.append({  
                'type': 'unlock_cliff',  
                'severity': 'high',  
                'description': f'Large unlock in month {month_data["month"]}'  
            })
```

```
(({month_data["new_unlocks"]}) tokens')  
})
```

return risks

SECTION 9: DISTRIBUTION BEST PRACTICES

Lessons from thousands of token launches:

Transparency Above All

Publish complete distribution information:

- Full allocation breakdown by category
- All investor terms and vesting schedules
- Wallet addresses for team and treasury
- Real-time tracking of unlocks and emissions

Hidden allocations destroy trust instantly when discovered. Better to explain controversial allocations upfront than face accusations of deception later.

Align Incentives Through Vesting

Everyone with significant allocation should have skin in the game for years:

- Team tokens: 4+ year vesting, 1 year cliff minimum
- Investor tokens: 2+ year vesting with cliff
- Advisor tokens: Match team vesting or close

Shorter vesting signals intent to dump. Longer vesting signals commitment.

Broad Distribution Wins

Distribute to many addresses, not few:

- Lower concentration reduces manipulation risk
- More holders means more advocates

- Broader distribution looks better to regulators
- Governance becomes more legitimate with diverse voters

Avoid giving single entities more than 5-10% of circulating supply outside of vesting contracts.

Reserve for Unknown Futures

Keep treasury allocation for things you can't predict:

- New chains requiring liquidity
- Partnership opportunities
- Emergency bug fixes requiring compensation
- Market conditions requiring intervention

Treasury should be governed by token holders, not controlled by team.

Monitor and Adapt

Distribution isn't set and forget:

- Track concentration metrics over time
- Adjust emission programs based on effectiveness
- Cut programs that aren't creating value
- Respond to market conditions appropriately

SECTION 10: DISTRIBUTION DOCUMENTATION

Every project should publish a Token Distribution Document covering:

Executive Summary

- Total supply and key allocations
- TGE circulating supply
- Major vesting milestones

Detailed Allocation Table

- Category, percentage, token amount, vesting terms
- For each investor round: price, amount raised, valuation

Vesting Schedule Visualization

- Chart showing unlock timeline
- Cumulative circulating supply over time

Contract Addresses

- All relevant token contracts
- Vesting contract addresses
- Treasury addresses

Governance

- How treasury is controlled
- How allocation changes are decided
- Multi-sig configurations

Example documentation structure:

```
def generate_distribution_document(project_data):
    doc = []

    doc.append('TOKEN DISTRIBUTION')
    doc.append('-' * 50)
    doc.append("")
    doc.append(f'Token: {project_data["name"]}
    ({project_data["symbol"]})')
    doc.append(f'Total Supply: {project_data["total_supply"]:,}')
    doc.append(f'TGE Circulating:
    {project_data["tge_circulating"]:,}
    ({project_data["tge_circulating"]/
    project_data["total_supply"]*100:.1f}%)')
    doc.append("")

    doc.append('ALLOCATION BREAKDOWN')
    doc.append('-' * 50)
```

```
for alloc in project_data['allocations']:
    doc.append(f'{alloc["category"]:20}
{alloc["percentage"]:5.1f}% {alloc["amount"]: > 15,} tokens')
    doc.append(f' Vesting: {alloc["vesting_description"]}')
    doc.append("")
```

```
doc.append('INVESTOR ROUNDS')
doc.append('-' * 50)
```

```
for round_info in project_data['rounds']:
    doc.append(f'{round_info["name"]}:')
    doc.append(f' Price: ${round_info["price"]}')
    doc.append(f' Raise: ${round_info["raise"]};,}')
    doc.append(f' Tokens: {round_info["tokens"]};,}')
    doc.append(f' Vesting: {round_info["vesting"]}')
    doc.append("")
```

```
doc.append('CONTRACT ADDRESSES')
doc.append('-' * 50)
```

```
for contract in project_data['contracts']:
    doc.append(f'{contract["name"]}: {contract["address"]}')

return '\n'.join(doc)
```

```
project = {
    'name': 'Example Token',
    'symbol': 'EXT',
    'total_supply': 1000000000,
    'tge_circulating': 150000000,
    'allocations': [
        {'category': 'Community', 'percentage': 40, 'amount':
400000000, 'vesting_description': 'Released over 4 years via
liquidity mining'},
        {'category': 'Team', 'percentage': 20, 'amount': 200000000,
'veesting_description': '4-year vesting, 1-year cliff'},
        {'category': 'Investors', 'percentage': 20, 'amount': 200000000,
```

```

'vesting_description': 'Varies by round'},
  {'category': 'Treasury', 'percentage': 15, 'amount': 150000000,
'vesting_description': 'Governance controlled'},
  {'category': 'Liquidity', 'percentage': 5, 'amount': 50000000,
'vesting_description': 'Immediate at TGE'}
],
'rounds': [
  {'name': 'Seed', 'price': 0.01, 'raise': 500000, 'tokens':
50000000, 'vesting': '2 years, 6 month cliff'},
  {'name': 'Private', 'price': 0.03, 'raise': 3000000, 'tokens':
100000000, 'vesting': '18 months, 3 month cliff'},
  {'name': 'Public', 'price': 0.05, 'raise': 2500000, 'tokens':
50000000, 'vesting': '6 months linear'}
],
'contracts': [
  {'name': 'Token', 'address': '0x1234...'},
  {'name': 'Vesting', 'address': '0x5678...'},
  {'name': 'Treasury', 'address': '0x9abc...'}
]
}

```

```

document = generate_distribution_document(project)
print(document)

```

Distribution determines destiny. The choices made about who gets tokens, when they get them, and under what conditions shape everything that follows. Get it right by prioritizing transparency, alignment, and broad participation. Your future stakeholders will thank you.

CHAPTER 8: BUILDING DEFI APPLICATIONS

Everything covered in this book converges here. Building real DeFi applications means integrating multiple protocols, handling edge cases gracefully, protecting users from attacks, and optimizing every aspect of the experience. This chapter shows how production DeFi applications work.

You've learned the individual pieces -- DEXes, lending, staking, stablecoins, tokenomics. Now we assemble them into cohesive applications that users trust with real money. The difference between tutorial code and production code is enormous. Production code handles failures, protects against exploits, optimizes gas costs, and provides clear feedback when things go wrong.

SECTION 1: COMPOSABILITY IN ACTION

DeFi's superpower is composability -- protocols that work together like Lego blocks. Your application can combine Uniswap swaps, Aave lending, and Curve stable swaps in a single transaction. No permissions needed. No business development deals. Just code calling code.

Understanding Composability

Traditional finance: Every integration requires contracts, APIs, business relationships, compliance reviews, and months of negotiation. Want to connect your bank to a broker? Good luck.

DeFi: Every protocol exposes public functions anyone can call. Want to swap tokens, deposit to lending, and stake the receipt in one transaction? Write the code.

This creates unprecedented innovation speed. New protocols compose existing ones. Yield aggregators combine dozens of strategies. Liquidation bots protect lending markets. MEV searchers arbitrage across venues instantaneously.

Multi-Protocol Integration Pattern

Here's a contract that demonstrates true composability -- flash loaning from Aave, swapping on Uniswap, and depositing to

Compound in a single transaction:

```
interface IPool {  
    function flashLoanSimple(  
        address receiverAddress,  
        address asset,  
        uint256 amount,  
        bytes calldata params,  
        uint16 referralCode  
    ) external;  
}
```

```
interface ISwapRouter {  
    struct ExactInputSingleParams {  
        address tokenIn;  
        address tokenOut;  
        uint24 fee;  
        address recipient;  
        uint256 amountIn;  
        uint256 amountOutMinimum;  
        uint160 sqrtPriceLimitX96;  
    }  
}
```

```
function exactInputSingle(ExactInputSingleParams calldata  
params) external returns (uint256 amountOut);  
}
```

```
interface ICToken {  
    function mint(uint256 mintAmount) external returns  
(uint256);  
    function redeem(uint256 redeemTokens) external returns  
(uint256);  
    function balanceOf(address owner) external view returns  
(uint256);  
}
```

```
interface IERC20 {
```

```
function approve(address spender, uint256 amount) external
returns (bool);
function transfer(address to, uint256 amount) external
returns (bool);
function balanceOf(address account) external view returns
(uint256);
}
```

```
contract ComposableDefiStrategy {
    address public owner;
    address public aavePool;
    address public uniswapRouter;

    uint256 constant PREMIUM_BPS = 9;

    constructor(address pool, address router) {
        owner = msg.sender;
        aavePool = pool;
        uniswapRouter = router;
    }
}
```

```
struct FlashParams {
    address tokenToSwap;
    address targetToken;
    uint24 poolFee;
    address cToken;
    uint256 minAmountOut;
}
```

```
function executeStrategy(
    address flashAsset,
    uint256 flashAmount,
    FlashParams calldata params
) external {
    require(msg.sender == owner);

    bytes memory data = abi.encode(params);
}
```

```

IPool(aavePool).flashLoanSimple(
address(this),
flashAsset,
flashAmount,
data,
0
);
}

```

```

function executeOperation(
address asset,
uint256 amount,
uint256 premium,
address initiator,
bytes calldata params
) external returns (bool) {
require(msg.sender == aavePool);
require(initiator == address(this));

```

```

FlashParams memory p = abi.decode(params,
(FlashParams));

```

```

IERC20(asset).approve(uniswapRouter, amount);

```

```

ISwapRouter.ExactInputSingleParams memory swapParams
= ISwapRouter.ExactInputSingleParams({
tokenIn: asset,
tokenOut: p.targetToken,
fee: p.poolFee,
recipient: address(this),
amountIn: amount,
amountOutMinimum: p.minAmountOut,
sqrtPriceLimitX96: 0
});

```

```

uint256 swapOutput =

```



```

ISwapRouter(uniswapRouter).exactInputSingle(swapParams);

IERC20(p.targetToken).approve(p.cToken, swapOutput);
ICToken(p.cToken).mint(swapOutput);

uint256 cTokenBalance =
ICToken(p.cToken).balanceOf(address(this));
ICToken(p.cToken).redeem(cTokenBalance);

uint256 amountOwed = amount + premium;

IERC20(asset).approve(aavePool, amountOwed);

return true;
}

function withdrawTokens(address token) external {
require(msg.sender == owner);
uint256 balance = IERC20(token).balanceOf(address(this));
IERC20(token).transfer(owner, balance);
}
}

```

This contract demonstrates the mental model: flash loan capital, use it across multiple protocols, return it with premium, keep any profit. Real strategies would include actual profit opportunities like arbitrage or liquidations.

JavaScript Integration Layer

The frontend needs to orchestrate these complex interactions:

```

import { ethers } from 'ethers'

class DefiComposer {
  constructor(provider) {
    this.provider = provider
  }
}

```

```
this.protocols = {}  
}
```

```
addProtocol(name, address, abi) {  
  this.protocols[name] = new ethers.Contract(address, abi,  
  this.provider)  
}
```

```
async simulateTransaction(txParams) {  
  try {  
    const result = await this.provider.call(txParams)  
    return { success: true, result }  
  } catch (error) {  
    return { success: false, error: error.message }  
  }  
}
```

```
async estimateGasWithBuffer(txParams, bufferPercent = 20)  
{  
  const estimate = await this.provider.estimateGas(txParams)  
  const buffer = estimate * BigInt(bufferPercent) / 100n  
  return estimate + buffer  
}
```

```
async buildMultiProtocolTx(steps) {  
  const calldata = []
```

```
  for (const step of steps) {  
    const protocol = this.protocols[step.protocol]  
    if (!protocol) {  
      throw new Error(`Unknown protocol: ${step.protocol}`)  
    }
```

```
    const encoded = protocol.interface.encodeFunctionData(  
      step.method,  
      step.params  
    )
```

```
calldata.push({  
  target: protocol.target,  
  callData: encoded,  
  value: step.value || 0n  
})  
}
```

```
return calldata  
}
```

```
async executeMulticall(multicallAddress, calls, signer) {  
  const multicall = new ethers.Contract(  
    multicallAddress,  
    ['function aggregate((address target, bytes callData)[] calls)  
    returns (uint256 blockNumber, bytes[] returnData)'],  
    signer  
  )
```

```
  const formattedCalls = calls.map(c => ({  
    target: c.target,  
    callData: c.callData  
  })))
```

```
  const tx = await multicall.aggregate(formattedCalls)  
  return tx.wait()  
}  
}
```

```
class YieldOptimizer {  
  constructor(composer) {  
    this.composer = composer  
  }
```

```
  async findBestYieldPath(tokenAddress, amount) {  
    const opportunities = await Promise.all([  
      this.checkAaveYield(tokenAddress, amount),
```

```
this.checkCompoundYield(tokenAddress, amount),  
this.checkCurveYield(tokenAddress, amount),  
this.checkConvexYield(tokenAddress, amount)  
])
```

```
return opportunities  
.filter(o => o.apy > 0)  
.sort((a, b) => b.apy - a.apy)  
}
```

```
async checkAaveYield(token, amount) {  
return { protocol: 'aave', apy: 0, risk: 'low' }  
}
```

```
async checkCompoundYield(token, amount) {  
return { protocol: 'compound', apy: 0, risk: 'low' }  
}
```

```
async checkCurveYield(token, amount) {  
return { protocol: 'curve', apy: 0, risk: 'medium' }  
}
```

```
async checkConvexYield(token, amount) {  
return { protocol: 'convex', apy: 0, risk: 'medium' }  
}
```

```
async executeYieldStrategy(signer, token, amount,  
strategyName) {  
const strategies = {  
'aave-supply': this.buildAaveSupply.bind(this),  
'compound-supply': this.buildCompoundSupply.bind(this),  
'curve-lp': this.buildCurveLP.bind(this),  
'convex-stake': this.buildConvexStake.bind(this)  
}
```

```
const builder = strategies[strategyName]  
if (!builder) {
```

```
throw new Error(`Unknown strategy: ${strategyName}`)  
}
```

```
const steps = await builder(token, amount)  
const calls = await  
this.composer.buildMultiProtocolTx(steps)
```

```
return calls  
}
```

```
async buildAaveSupply(token, amount) {  
  return [  
    {  
      protocol: 'aave',  
      method: 'supply',  
      params: [token, amount, this.composer.provider.address, 0]  
    }  
  ]  
}
```

```
async buildCompoundSupply(token, amount) {  
  return [  
    {  
      protocol: 'compound',  
      method: 'mint',  
      params: [amount]  
    }  
  ]  
}
```

```
async buildCurveLP(token, amount) {  
  return []  
}
```

```
async buildConvexStake(token, amount) {  
  return []  
}
```

}

SECTION 2: PROTECTING AGAINST SANDWICH ATTACKS

Sandwich attacks are the most common MEV attack facing DeFi users. An attacker sees your pending swap, places a buy order before yours (frontrun), lets your trade execute at a worse price, then sells immediately after (backrun) for profit. Your loss is their gain.

Understanding the Attack

1. You submit: Buy 10 ETH worth of TOKEN at market price
2. Attacker sees your pending tx in mempool
3. Attacker frontruns: Buys TOKEN, pushing price up
4. Your tx executes: You buy at inflated price
5. Attacker backruns: Sells TOKEN at the price you pushed it to
6. Result: You got fewer tokens, attacker profits

Slippage Protection

The primary defense is setting appropriate slippage tolerance -- the maximum price difference you'll accept:

```
class SlippageProtector {  
  constructor(defaultSlippageBps = 50) {  
    this.defaultSlippageBps = defaultSlippageBps  
  }  
  
  calculateMinOutput(expectedOutput, slippageBps) {  
    const slippage = slippageBps || this.defaultSlippageBps  
    const minOutput = expectedOutput * (10000n -  
      BigInt(slippage)) / 10000n  
    return minOutput  
  }  
}
```

```
calculateMaxInput(expectedInput, slippageBps) {  
  const slippage = slippageBps || this.defaultSlippageBps  
  const maxInput = expectedInput * (10000n +  
    BigInt(slippage)) / 10000n  
  return maxInput  
}
```

```
recommendSlippage(tokenInfo) {  
  if (tokenInfo.isStablePair) {  
    return 10  
  }
```

```
  if (tokenInfo.liquidityUSD > 10000000) {  
    return 50  
  }
```

```
  if (tokenInfo.liquidityUSD > 1000000) {  
    return 100  
  }
```

```
  if (tokenInfo.liquidityUSD > 100000) {  
    return 200  
  }
```

```
  return 500  
}
```

```
validateSlippage(slippageBps) {  
  if (slippageBps < 1) {  
    return {  
      valid: false,  
      warning: 'Slippage too low, transaction likely to fail'  
    }  
  }  
}
```

```
  if (slippageBps > 1000) {
```

```

return {
  valid: false,
  warning: 'Slippage too high, vulnerable to sandwich attacks'
}
}

if (slippageBps > 500) {
  return {
    valid: true,
    warning: 'High slippage may result in unfavorable execution'
  }
}

return { valid: true, warning: null }
}
}

```

Private Transaction Services

The best protection is hiding your transaction from public mempools. Services like Flashbots allow private transaction submission:

```

import { ethers } from 'ethers'

class FlashbotsProtection {
  constructor(provider, authSigner) {
    this.provider = provider
    this.authSigner = authSigner
    this.flashbotsRpc = 'https://rpc.flashbots.net'
  }

  async sendPrivateTransaction(signedTx, maxBlockNumber) {
    const payload = {
      jsonrpc: '2.0',
      id: 1,
      method: 'eth_sendPrivateTransaction',

```



```
params: [{  
  tx: signedTx,  
  maxBlockNumber: ethers.toBeHex(maxBlockNumber)  
}]  
}
```

```
const response = await fetch(this.flashbotsRpc, {  
  method: 'POST',  
  body: JSON.stringify(payload),  
  headers: {  
    'Content-Type': 'application/json',  
    'X-Flashbots-Signature': await this.signPayload(payload)  
  }  
})
```

```
return response.json()  
}
```

```
async signPayload(payload) {  
  const body = JSON.stringify(payload)  
  const signature = await  
this.authSigner.signMessage(ethers.id(body))  
  return `${this.authSigner.address}:${signature}`  
}
```

```
async checkPrivateTxStatus(txHash) {  
  const payload = {  
    jsonrpc: '2.0',  
    id: 1,  
    method: 'eth_getPrivateTransactionStatus',  
    params: [txHash]  
  }  
}
```

```
const response = await fetch(this.flashbotsRpc, {  
  method: 'POST',  
  body: JSON.stringify(payload),  
  headers: { 'Content-Type': 'application/json' }  
})
```

```
}}
```

```
return response.json()  
}  
}
```

```
class MEVProtectedSwap {  
  constructor(provider, flashbots) {  
    this.provider = provider  
    this.flashbots = flashbots  
    this.slippageProtector = new SlippageProtector()  
  }
```

```
  async executeProtectedSwap(signer, swapParams) {  
    const quote = await this.getQuote(swapParams)
```

```
    const minOutput =  
    this.slippageProtector.calculateMinOutput(  
      quote.expectedOutput,  
      swapParams.slippageBps  
    )
```

```
    const tx = await this.buildSwapTransaction(  
      swapParams,  
      minOutput,  
      signer  
    )
```

```
    const signedTx = await signer.signTransaction(tx)
```

```
    const currentBlock = await this.provider.getBlockNumber()  
    const maxBlock = currentBlock + 25
```

```
    const result = await this.flashbots.sendPrivateTransaction(  
      signedTx,  
      maxBlock  
    )
```

```
return result
}
```

```
async getQuote(swapParams) {
return { expectedOutput: 0n }
}
```

```
async buildSwapTransaction(swapParams, minOutput, signer)
{
return {}
}
}
```

On-Chain Sandwich Protection

Some protocols implement on-chain protections:

```
contract SandwichResistantSwap {
mapping(bytes32 => uint256) public commitments;
mapping(bytes32 => bool) public revealed;
```

```
uint256 public commitDelay;
uint256 public commitExpiry;
```

```
address public router;
```

```
constructor(address swapRouter, uint256 delay, uint256
expiry) {
router = swapRouter;
commitDelay = delay;
commitExpiry = expiry;
}
```

```
function commitSwap(bytes32 commitHash) external {
commitments[commitHash] = block.number;
}
```

```

function revealAndSwap(
    address tokenIn,
    address tokenOut,
    uint256 amountIn,
    uint256 minAmountOut,
    bytes32 secret
) external {
    bytes32 commitHash = keccak256(abi.encodePacked(
        msg.sender,
        tokenIn,
        tokenOut,
        amountIn,
        minAmountOut,
        secret
    ));

    uint256 commitBlock = commitments[commitHash];
    require(commitBlock > 0);
    require(block.number >= commitBlock + commitDelay);
    require(block.number <= commitBlock + commitExpiry);
    require(!revealed[commitHash]);

    revealed[commitHash] = true;

    IERC20(tokenIn).transferFrom(msg.sender, address(this),
    amountIn);
    IERC20(tokenIn).approve(router, amountIn);

    bytes memory swapData = abi.encodeWithSignature(
        "swap(address,address,uint256,uint256,address)",
        tokenIn,
        tokenOut,
        amountIn,
        minAmountOut,
        msg.sender
    );

```

```

    (bool success, ) = router.call(swapData);
    require(success);
  }
}

```

This commit-reveal scheme prevents sandwiching because attackers can't see swap details until after they commit. The delay ensures transactions are ordered before details are known.

SECTION 3: SLIPPAGE HANDLING

Slippage occurs when execution price differs from quoted price. Handling it well means happy users. Handling it poorly means failed transactions and frustration.

Dynamic Slippage Calculation

```

import { ethers } from 'ethers'

class DynamicSlippageCalculator {
  constructor(provider) {
    this.provider = provider
  }

  async calculateOptimalSlippage(params) {
    const volatility = await this.getVolatility(params.tokenA,
    params.tokenB)
    const liquidity = await this.getLiquidity(params.pool)
    const tradeSize = params.amountIn
    const priceImpact = await this.estimatePriceImpact(params)

    let baseSlippage = 30

    if (volatility > 0.1) {

```

```
baseSlippage += 50
} else if (volatility > 0.05) {
baseSlippage += 25
}
```

```
if (priceImpact > 300) {
baseSlippage += 100
} else if (priceImpact > 100) {
baseSlippage += 50
}
```

```
const gasPrice = await this.provider.getFeeData()
const baseGwei = Number(gasPrice.gasPrice) / 1e9
```

```
if (baseGwei > 100) {
baseSlippage += 20
} else if (baseGwei > 50) {
baseSlippage += 10
}
```

```
return Math.min(baseSlippage, 500)
}
```

```
async getVolatility(tokenA, tokenB) {
return 0.03
}
```

```
async getLiquidity(pool) {
return 1000000
}
```

```
async estimatePriceImpact(params) {
return 50
}
```

```
async simulateWithSlippage(swapParams, slippages) {
const results = []
```

```

for (const slippage of slippages) {
  const minOutput = this.calculateMinOutput(
    swapParams.expectedOutput,
    slippage
  )

  const simulation = await this.simulateSwap({
    ...swapParams,
    minAmountOut: minOutput
  })

  results.push({
    slippage,
    minOutput: ethers.formatEther(minOutput),
    expectedSuccess: simulation.success,
    reason: simulation.reason
  })
}

return results
}

calculateMinOutput(expected, slippageBps) {
  return expected * (10000n - BigInt(slippageBps)) / 10000n
}

async simulateSwap(params) {
  return { success: true, reason: null }
}
}

```

Price Impact Warnings

Users need clear warnings when trades have significant impact:

```

class PriceImpactAnalyzer {
  constructor() {
    this.warningThresholds = {
      low: 100,
      medium: 300,
      high: 500,
      extreme: 1000
    }
  }

  calculatePriceImpact(amountIn, amountOut, spotPrice) {
    const expectedOut = amountIn * spotPrice
    const actualOut = amountOut

    if (expectedOut === 0n) return 0

    const impact = ((expectedOut - actualOut) * 10000n) /
    expectedOut
    return Number(impact)
  }

  getPriceImpactWarning(impactBps) {
    if (impactBps < this.warningThresholds.low) {
      return {
        level: 'none',
        message: null,
        color: 'green'
      }
    }

    if (impactBps < this.warningThresholds.medium) {
      return {
        level: 'low',
        message: `Price impact: ${((impactBps / 100).toFixed(2))}%`,
        color: 'yellow'
      }
    }
  }
}

```



```
if (impactBps < this.warningThresholds.high) {  
  return {  
    level: 'medium',  
    message: `Warning: ${((impactBps / 100).toFixed(2))}% price  
    impact. Consider splitting into smaller trades.`,  
    color: 'orange'  
  }  
}
```

```
if (impactBps < this.warningThresholds.extreme) {  
  return {  
    level: 'high',  
    message: `High price impact: ${((impactBps /  
100).toFixed(2))}%. You will lose significant value.`,  
    color: 'red'  
  }  
}
```

```
return {  
  level: 'extreme',  
  message: `EXTREME price impact: ${((impactBps /  
100).toFixed(2))}%. This trade is not recommended.`,  
  color: 'red',  
  requireConfirmation: true  
}  
}
```

```
suggestSplitTrade(totalAmount, targetImpactBps,  
poolLiquidity) {  
  const trades = []  
  let remaining = totalAmount
```

```
  while (remaining > 0n) {  
    const tradeSize = this.calculateMaxTradeSize(  
      remaining,  
      targetImpactBps,
```

```

poolLiquidity
)

trades.push(tradeSize)
remaining -= tradeSize
}

return trades
}

calculateMaxTradeSize(amount, targetImpact, liquidity) {
return amount
}
}

```

SECTION 4: GAS OPTIMIZATION

Every wei saved on gas is value preserved for users.
Production applications obsess over gas efficiency.

Contract-Level Optimizations

```

contract GasOptimizedVault {
mapping(address => uint256) private _balances;
mapping(address => mapping(address => uint256))
private _allowances;

uint256 private _totalSupply;
address public immutable asset;
address public immutable rewardToken;

uint256 private constant PRECISION = 1e18;

uint256 public rewardPerTokenStored;
uint256 public lastUpdateTime;
uint256 public rewardRate;

```

```

mapping(address => uint256) public
userRewardPerTokenPaid;
mapping(address => uint256) public rewards;

constructor(address assetAddress, address rewardAddress) {
    asset = assetAddress;
    rewardToken = rewardAddress;
}

function deposit(uint256 amount) external {
    _updateReward(msg.sender);

    IERC20(asset).transferFrom(msg.sender, address(this),
amount);

    unchecked {
        _balances[msg.sender] += amount;
        _totalSupply += amount;
    }
}

function withdraw(uint256 amount) external {
    _updateReward(msg.sender);

    unchecked {
        _balances[msg.sender] -= amount;
        _totalSupply -= amount;
    }

    IERC20(asset).transfer(msg.sender, amount);
}

function depositAndStake(uint256 amount, address
stakingPool) external {
    IERC20(asset).transferFrom(msg.sender, address(this),
amount);

```

```
IERC20(asset).approve(stakingPool, amount);
IStaking(stakingPool).stakeFor(msg.sender, amount);
}
```

```
function batchClaim(address[] calldata users) external {
    uint256 length = users.length;
```

```
    for (uint256 i; i < length; ) {
        address user = users[i];
        _updateReward(user);
```

```
        uint256 reward = rewards[user];
        if (reward > 0) {
            rewards[user] = 0;
            IERC20(rewardToken).transfer(user, reward);
        }
```

```
        unchecked { ++i; }
    }
}
```

```
function _updateReward(address account) private {
    rewardPerTokenStored = _rewardPerToken();
    lastUpdateTime = block.timestamp;
```

```
    if (account != address(0)) {
        rewards[account] = _earned(account);
        userRewardPerTokenPaid[account] =
            rewardPerTokenStored;
    }
}
```

```
function _rewardPerToken() private view returns (uint256) {
    if (_totalSupply == 0) {
        return rewardPerTokenStored;
    }
```

```

return rewardPerTokenStored + (
    (block.timestamp - lastUpdateTime) * rewardRate *
    PRECISION / _totalSupply
);
}

```

```

function _earned(address account) private view returns
(uint256) {
    return (
        _balances[account] * (_rewardPerToken() -
        userRewardPerTokenPaid[account]) / PRECISION
    ) + rewards[account];
}

```

```

function balanceOf(address account) external view returns
(uint256) {
    return _balances[account];
}

```

```

function totalSupply() external view returns (uint256) {
    return _totalSupply;
}
}

```

```

interface IStaking {
    function stakeFor(address user, uint256 amount) external;
}

```

Key optimizations applied:

- Immutable variables for constants (saves SLOAD)
- Unchecked arithmetic where overflow impossible (saves gas)
- Packed storage where possible
- Batch operations reduce per-call overhead
- Calldata instead of memory for read-only arrays
- Pre-increment (+ + i) slightly cheaper than post-increment

Frontend Gas Estimation

```

class GasOptimizer {
  constructor(provider) {
    this.provider = provider
    this.gasHistoryBlocks = 20
  }

  async estimateOptimalGas(tx) {
    const [baseFee, priorityFee, gasLimit] = await Promise.all([
      this.getBaseFee(),
      this.getOptimalPriorityFee(),
      this.estimateGasLimit(tx)
    ])

    return {
      maxFeePerGas: baseFee * 2n + priorityFee,
      maxPriorityFeePerGas: priorityFee,
      gasLimit: gasLimit * 120n / 100n
    }
  }

  async getBaseFee() {
    const block = await this.provider.getBlock('latest')
    return block.baseFeePerGas || 0n
  }

  async getOptimalPriorityFee() {
    const feeHistory = await this.provider.send('eth_feeHistory', [
      this.gasHistoryBlocks,
      'latest',
      [25, 50, 75]
    ])

    const recentPriorities = feeHistory.reward
      .map(r => BigInt(r[1]))
      .slice(-10)
  }
}

```

```
const sum = recentPriorities.reduce((a, b) => a + b, 0n)  
return sum / BigInt(recentPriorities.length)  
}
```

```
async estimateGasLimit(tx) {  
  try {  
    return await this.provider.estimateGas(tx)  
  } catch (error) {  
    return 500000n  
  }  
}
```

```
async getGasCostInUSD(gasLimit, maxFeePerGas, ethPrice) {  
  const gasCostWei = gasLimit * maxFeePerGas  
  const gasCostEth = Number(gasCostWei) / 1e18  
  return gasCostEth * ethPrice  
}
```

```
async shouldWaitForLowerGas(currentGas, urgency) {  
  const historicalAverage = await  
  this.getHistoricalAverageGas()
```

```
  if (urgency === 'immediate') {  
    return false  
  }
```

```
  if (urgency === 'normal') {  
    return currentGas > historicalAverage * 150n / 100n  
  }
```

```
  if (urgency === 'low') {  
    return currentGas > historicalAverage  
  }
```

```
  return false  
}
```

```

async getHistoricalAverageGas() {
  const feeHistory = await this.provider.send('eth_feeHistory', [
    100,
    'latest',
    []
  ])

  const baseFees = feeHistory.baseFeePerGas.map(f =>
    BigInt(f))
  const sum = baseFees.reduce((a, b) => a + b, 0n)
  return sum / BigInt(baseFees.length)
}

formatGasPrice(weiAmount) {
  const gwei = Number(weiAmount) / 1e9
  return `${gwei.toFixed(2)} gwei`
}
}

```

SECTION 5: ERROR HANDLING AND RECOVERY

DeFi transactions fail. Networks congest. Prices move. Contracts revert. Robust applications handle failures gracefully.

Transaction Error Classification

```

class TransactionErrorHandler {
  constructor() {
    this.errorPatterns = {
      'insufficient funds': {
        type: 'USER_BALANCE',
        userMessage: 'Insufficient balance to complete transaction',
        recoverable: false
      },
      'execution reverted': {

```



```
type: 'CONTRACT_REVERT',
userMessage: 'Transaction would fail on-chain',
recoverable: true
},
'nonce too low': {
type: 'NONCE_ERROR',
userMessage: 'Transaction conflict detected',
recoverable: true
},
'replacement fee too low': {
type: 'GAS_ERROR',
userMessage: 'Gas price too low to replace pending
transaction',
recoverable: true
},
'timeout': {
type: 'NETWORK_ERROR',
userMessage: 'Network request timed out',
recoverable: true
},
'INSUFFICIENT_OUTPUT_AMOUNT': {
type: 'SLIPPAGE',
userMessage: 'Price moved beyond slippage tolerance',
recoverable: true
},
'EXPIRED': {
type: 'DEADLINE',
userMessage: 'Transaction deadline expired',
recoverable: true
},
'STF': {
type: 'TRANSFER_FAILED',
userMessage: 'Token transfer failed',
recoverable: false
}
}
}
```

```

classifyError(error) {
  const errorString = error.message || error.toString()

  for (const [pattern, classification] of
Object.entries(this.errorPatterns)) {
    if
(errorString.toLowerCase().includes(pattern.toLowerCase())) {
      return {
        ...classification,
        originalError: errorString
      }
    }
  }

  return {
    type: 'UNKNOWN',
    userMessage: 'An unexpected error occurred',
    recoverable: false,
    originalError: errorString
  }
}

getSuggestion(errorType) {
  const suggestions = {
    'USER_BALANCE': 'Please add funds to your wallet',
    'CONTRACT_REVERT': 'Try adjusting your slippage tolerance
or amount',
    'NONCE_ERROR': 'Wait for pending transactions to confirm',
    'GAS_ERROR': 'Increase gas price or wait for pending
transactions',
    'NETWORK_ERROR': 'Check your internet connection and try
again',
    'SLIPPAGE': 'Increase slippage tolerance or reduce trade size',
    'DEADLINE': 'Submit transaction again with new deadline',
    'TRANSFER_FAILED': 'Ensure you have approved the token for
spending'
  }
}

```

```
}
```

```
return suggestions[errorType] || 'Please try again or contact support'
```

```
}
```

```
}
```

```
class TransactionRetrier {
```

```
  constructor(provider, maxRetries = 3) {
```

```
    this.provider = provider
```

```
    this.maxRetries = maxRetries
```

```
    this.errorHandler = new TransactionErrorHandler()
```

```
  }
```

```
  async executeWithRetry(buildTx, options = {}) {
```

```
    let lastError
```

```
    let attempts = 0
```

```
    while (attempts < this.maxRetries) {
```

```
      try {
```

```
        const tx = await buildTx(attempts)
```

```
        const receipt = await tx.wait()
```

```
      return {
```

```
        success: true,
```

```
        receipt,
```

```
        attempts: attempts + 1
```

```
      }
```

```
    } catch (error) {
```

```
      lastError = error
```

```
      const classified = this.errorHandler.classifyError(error)
```

```
      if (!classified.recoverable) {
```

```
        throw error
```

```
      }
```

```
      attempts ++
```

```

if (attempts < this.maxRetries) {
  const delay = this.calculateBackoff(attempts)
  await this.sleep(delay)
}
}
}

return {
  success: false,
  error: lastError,
  attempts,
  suggestion: this.errorHandler.getSuggestion(
    this.errorHandler.classifyError(lastError).type
  )
}
}

calculateBackoff(attempt) {
  return Math.min(1000 * Math.pow(2, attempt), 30000)
}

sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms))
}
}

```

Stuck Transaction Recovery

```

class TransactionRecovery {
  constructor(provider, signer) {
    this.provider = provider
    this.signer = signer
  }

  async getStuckTransactions(address) {
    const nonce = await

```

```

this.provider.getTransactionCount(address, 'latest')
const pendingNonce = await
this.provider.getTransactionCount(address, 'pending')

if (pendingNonce > nonce) {
  return {
    hasStuckTx: true,
    confirmedNonce: nonce,
    pendingNonce: pendingNonce,
    stuckCount: pendingNonce - nonce
  }
}

return { hasStuckTx: false }
}

async speedUpTransaction(txHash, gasPriceMultiplier = 1.2)
{
  const tx = await this.provider.getTransaction(txHash)

  if (!tx || tx.blockNumber) {
    throw new Error('Transaction already confirmed or not
found')
  }

  const newMaxFee = tx.maxFeePerGas
  ? tx.maxFeePerGas * BigInt(Math.floor(gasPriceMultiplier *
100)) / 100n
  : tx.gasPrice * BigInt(Math.floor(gasPriceMultiplier * 100)) /
100n

  const newMaxPriority = tx.maxPriorityFeePerGas
  ? tx.maxPriorityFeePerGas *
  BigInt(Math.floor(gasPriceMultiplier * 100)) / 100n
  : null

  const replacementTx = {

```

```
to: tx.to,  
value: tx.value,  
data: tx.data,  
nonce: tx.nonce,  
gasLimit: tx.gasLimit,  
maxFeePerGas: newMaxFee,  
maxPriorityFeePerGas: newMaxPriority || newMaxFee  
}
```

```
return this.signer.sendTransaction(replacementTx)  
}
```

```
async cancelTransaction(nonce, gasPrice) {  
  const cancelTx = {  
    to: await this.signer.getAddress(),  
    value: 0n,  
    data: '0x',  
    nonce: nonce,  
    maxFeePerGas: gasPrice,  
    maxPriorityFeePerGas: gasPrice  
  }  
}
```

```
return this.signer.sendTransaction(cancelTx)  
}
```

```
async clearAllPendingTransactions() {  
  const address = await this.signer.getAddress()  
  const stuckInfo = await this.getStuckTransactions(address)  
  
  if (!stuckInfo.hasStuckTx) {  
    return { cleared: 0 }  
  }  
}
```

```
const feeData = await this.provider.getFeeData()  
const highGas = feeData.maxFeePerGas * 2n
```

```
const receipts = []
```

```

    for (let nonce = stuckInfo.confirmedNonce; nonce <
stuckInfo.pendingNonce; nonce + +) {
      const tx = await this.cancelTransaction(nonce, highGas)
      const receipt = await tx.wait()
      receipts.push(receipt)
    }

    return { cleared: receipts.length, receipts }
  }
}

```

SECTION 6: MULTI-CHAIN CONSIDERATIONS

Production DeFi applications span multiple chains. Each chain has different characteristics:

```

class MultiChainManager {
  constructor() {
    this.chains = new Map()
  }

  addChain(chainId, config) {
    this.chains.set(chainId, {
      provider: config.provider,
      contracts: config.contracts,
      blockTime: config.blockTime,
      confirmations: config.confirmations,
      gasConfig: config.gasConfig
    })
  }

  async getProvider(chainId) {
    const chain = this.chains.get(chainId)
    if (!chain) {
      throw new Error(`Chain ${chainId} not configured`)
    }
  }
}

```

```
}  
return chain.provider  
}
```

```
async executeOnChain(chainId, action) {  
  const chain = this.chains.get(chainId)  
  if (!chain) {  
    throw new Error(`Chain ${chainId} not configured`)  
  }
```

```
  const provider = chain.provider  
  return action(provider, chain.contracts)  
}
```

```
async waitForConfirmations(chainId, txHash) {  
  const chain = this.chains.get(chainId)  
  const receipt = await  
chain.provider.getTransactionReceipt(txHash)
```

```
  if (!receipt) {  
    throw new Error("Transaction not found")  
  }
```

```
  const currentBlock = await chain.provider.getBlockNumber()  
  const confirmations = currentBlock - receipt.blockNumber
```

```
  return {  
    confirmed: confirmations >= chain.confirmations,  
    currentConfirmations: confirmations,  
    requiredConfirmations: chain.confirmations  
  }  
}
```

```
getGasConfig(chainId) {  
  const chain = this.chains.get(chainId)  
  return chain?.gasConfig || {  
    maxGwei: 100,
```



```
priorityGwei: 2
}
}
}
```

```
class ChainAbstractionLayer {
  constructor(multiChainManager) {
    this.manager = multiChainManager

    this.tokenMappings = {
      'USDC': {
        1: '0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48',
        137: '0x2791Bca1f2de4661ED88A30C99A7a9449Aa84174',
        42161:
'0xFF970A61A04b1cA14834A43f5dE4533eBDDb5CC8',
        10: '0x7F5c764cBc14f9669B88837ca1490cCa17c31607'
      },
      'WETH': {
        1: '0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2',
        137: '0x7ceB23fD6bC0adD59E62ac25578270cFf1b9f619',
        42161: '0x82aF49447D8a07e3bd95BD0d56f35241523fBab1',
        10: '0x4200000000000000000000000000000000000000000000000000000000000006'
      }
    }

    this.dexMappings = {
      1: {
        uniswap:
'0xE592427A0AEce92De3Edee1F18E0157C05861564',
        sushiswap:
'0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F'
      },
      137: {
        quickswap:
'0xa5E0829CaCEd8fFDD4De3c43696c57F7D7A678ff',
        sushiswap:
'0x1b02dA8Cb0d097eB8D57A175b88c7D8b47997506'
```

```

},
42161: {
  uniswap:
'0xE592427A0AEce92De3Edee1F18E0157C05861564',
  sushiswap:
'0x1b02dA8Cb0d097eB8D57A175b88c7D8b47997506'
}
}
}
}

```

```

getTokenAddress(symbol, chainId) {
  const mapping = this.tokenMappings[symbol]
  if (!mapping || !mapping[chainId]) {
    throw new Error(`Token ${symbol} not available on chain
    ${chainId}`)
  }
  return mapping[chainId]
}

```

```

getDexRouter(chainId, dexName) {
  const chainDexes = this.dexMappings[chainId]
  if (!chainDexes) {
    throw new Error(`No DEX configured for chain ${chainId}`)
  }

```

```

  const router = chainDexes[dexName] ||
  Object.values(chainDexes)[0]
  return router
}

```

```

async findBestChainForSwap(tokenIn, tokenOut, amount) {
  const results = []

```

```

  for (const [chainId, config] of this.manager.chains) {
    try {
      const tokenInAddr = this.getTokenAddress(tokenIn, chainId)
      const tokenOutAddr = this.getTokenAddress(tokenOut,

```

chainId)

```
const quote = await this.getQuote(chainId, tokenInAddr,  
tokenOutAddr, amount)
```

```
const gasCost = await this.estimateGasCost(chainId)
```

```
results.push({  
  chainId,  
  quote,  
  gasCost,  
  netValue: quote - gasCost  
})
```

```
} catch (e) {  
  continue  
}  
}
```

```
return results.sort((a, b) => b.netValue - a.netValue)  
}
```

```
async getQuote(chainId, tokenIn, tokenOut, amount) {  
  return 0n  
}
```

```
async estimateGasCost(chainId) {  
  return 0n  
}  
}
```

SECTION 7: PRODUCTION MONITORING

Production applications require extensive monitoring:

```
class DefiApplicationMonitor {  
  constructor(config) {  
    this.provider = config.provider
```

```
this.contracts = config.contracts
this.alertCallback = config.onAlert
}
```

```
async monitorContractHealth() {
  const results = {}
```

```
  for (const [name, address] of Object.entries(this.contracts)) {
    const code = await this.provider.getCode(address)
```

```
    if (code === '0x') {
      this.alert('critical', `Contract ${name} has no code at
      ${address}`)
      results[name] = { status: 'error', message: 'No code' }
    } else {
      results[name] = { status: 'healthy' }
    }
  }
}
```

```
  return results
}
```

```
async monitorPriceDeviation(priceFeeds,
maxDeviationPercent) {
  for (const feed of priceFeeds) {
    const onChainPrice = await
    this.getOnChainPrice(feed.address)
    const offChainPrice = await
    this.getOffChainPrice(feed.symbol)
```

```
    const deviation = Math.abs(onChainPrice - offChainPrice) /
    offChainPrice * 100
```

```
    if (deviation > maxDeviationPercent) {
      this.alert('warning', `Price deviation for ${feed.symbol}:
      ${deviation.toFixed(2)}%`)
    }
  }
}
```

```
}  
}
```

```
async monitorLiquidity(pools, minLiquidityUSD) {  
  for (const pool of pools) {  
    const liquidity = await this.getPoolLiquidity(pool.address)  
  
    if (liquidity < minLiquidityUSD) {  
      this.alert('warning', `Low liquidity in ${pool.name}: $  
${liquidity.toFixed(2)}`)  
    }  
  }  
}
```

```
async monitorGasSpikes(maxGweiThreshold) {  
  const feeData = await this.provider.getFeeData()  
  const currentGwei = Number(feeData.gasPrice) / 1e9  
  
  if (currentGwei > maxGweiThreshold) {  
    this.alert('info', `High gas prices: ${currentGwei.toFixed(2)}  
gwei`)  
  }  
}
```

```
return currentGwei  
}
```

```
async getOnChainPrice(address) {  
  return 0  
}
```

```
async getOffChainPrice(symbol) {  
  return 0  
}
```

```
async getPoolLiquidity(address) {  
  return 0  
}
```

```
alert(level, message) {  
  const alert = {  
    level,  
    message,  
    timestamp: new Date().toISOString()  
  }
```

```
  console.log(`[${level.toUpperCase()}] ${message}`)
```

```
  if (this.alertCallback) {  
    this.alertCallback(alert)  
  }  
}  
}
```

```
class TransactionAnalytics {  
  constructor() {  
    this.transactions = []  
  }
```

```
  recordTransaction(tx) {  
    this.transactions.push({  
      hash: tx.hash,  
      from: tx.from,  
      to: tx.to,  
      value: tx.value,  
      gasUsed: tx.gasUsed,  
      gasPrice: tx.effectiveGasPrice,  
      timestamp: Date.now(),  
      status: tx.status  
    })  
  }  
}
```

```
  getStats(timeframeMs = 24 * 60 * 60 * 1000) {  
    const cutoff = Date.now() - timeframeMs  
    const recent = this.transactions.filter(tx => tx.timestamp >
```

cutoff)

```
if (recent.length === 0) {  
  return { count: 0 }  
}
```

```
const totalGas = recent.reduce((sum, tx) =>  
  sum + BigInt(tx.gasUsed) * BigInt(tx.gasPrice), 0n  
)
```

```
const successCount = recent.filter(tx => tx.status ===  
1).length
```

```
return {  
  count: recent.length,  
  successRate: (successCount / recent.length * 100).toFixed(2),  
  totalGasSpentWei: totalGas.toString(),  
  totalGasSpentEth: (Number(totalGas) / 1e18).toFixed(6),  
  averageGasPerTx: (Number(totalGas) / recent.length /  
1e18).toFixed(8)  
}  
}  
}
```

SECTION 8: TESTING DEFI APPLICATIONS

Testing DeFi code requires specialized approaches:

Forking Mainnet

```
const { ethers } = require('hardhat')
```

```
async function setupMainnetFork() {  
  await network.provider.request({  
    method: 'hardhat_reset',  
    params: [{
```

```

forking: {
  jsonRpcUrl: process.env.ETH_RPC_URL,
  blockNumber: 18000000
}
}]
})
}

```

```

async function impersonateAccount(address) {
  await network.provider.request({
    method: 'hardhat_impersonateAccount',
    params: [address]
  })

  return ethers.getSigner(address)
}

```

```

async function setBalance(address, balance) {
  await network.provider.send('hardhat_setBalance', [
    address,
    ethers.toBeHex(balance)
  ])
}

```

```

async function testSwapOnFork() {
  await setupMainnetFork()

  const usdcWhale =
'0x47ac0Fb4F2D84898e4D9E7b4DaB3C24507a6D503'
  const whaleSigner = await impersonateAccount(usdcWhale)
  await setBalance(usdcWhale, ethers.parseEther('10'))

  const usdc = await ethers.getContractAt(
    'IERC20',
    '0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48',
    whaleSigner
  )
}

```



```

const router = await ethers.getContractAt(
  'ISwapRouter',
  '0xE592427A0AEce92De3Edee1F18E0157C05861564',
  whaleSigner
)

const amountIn = 10000n * 10n ** 6n

await usdc.approve(router.target, amountIn)

const params = {
  tokenIn: usdc.target,
  tokenOut:
'0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2',
  fee: 3000,
  recipient: usdcWhale,
  amountIn: amountIn,
  amountOutMinimum: 0,
  sqrtPriceLimitX96: 0
}

const tx = await router.exactInputSingle(params)
const receipt = await tx.wait()

console.log('Swap executed:', receipt.hash)
}

```

Integration Test Patterns

```

const { expect } = require('chai')
const { ethers } = require('hardhat')

describe('DeFi Integration Tests', function() {
  let vault
  let token
  let owner

```

```
let user
```

```
beforeEach(async function() {  
  [owner, user] = await ethers.getSigners()
```

```
  const Token = await  
  ethers.getContractFactory('MockERC20')  
  token = await Token.deploy('Test Token', 'TEST', 18)
```

```
  const Vault = await  
  ethers.getContractFactory('GasOptimizedVault')  
  vault = await Vault.deploy(token.target, token.target)
```

```
  await token.mint(user.address, ethers.parseEther('10000'))  
  await token.connect(user).approve(vault.target,  
  ethers.MaxUint256)  
})
```

```
it('should accept deposits', async function() {  
  const depositAmount = ethers.parseEther('1000')
```

```
  await vault.connect(user).deposit(depositAmount)
```

```
  expect(await  
  vault.balanceOf(user.address)).to.equal(depositAmount)  
  expect(await vault.totalSupply()).to.equal(depositAmount)  
})
```

```
it('should handle withdrawals correctly', async function() {  
  const depositAmount = ethers.parseEther('1000')  
  await vault.connect(user).deposit(depositAmount)
```

```
  const withdrawAmount = ethers.parseEther('500')  
  await vault.connect(user).withdraw(withdrawAmount)
```

```
  expect(await  
  vault.balanceOf(user.address)).to.equal(depositAmount -
```

```
withdrawAmount)  
  })
```

```
it('should revert on insufficient balance', async function() {  
  const depositAmount = ethers.parseEther('1000')  
  await vault.connect(user).deposit(depositAmount)
```

```
  const withdrawAmount = ethers.parseEther('1500')
```

```
  await expect(  
    vault.connect(user).withdraw(withdrawAmount)  
  ).to.be.reverted  
})
```

```
it('should handle multiple users', async function() {  
  const [, , user2] = await ethers.getSigners()
```

```
  await token.mint(user2.address, ethers.parseEther('10000'))  
  await token.connect(user2).approve(vault.target,  
    ethers.MaxUint256)
```

```
  await vault.connect(user).deposit(ethers.parseEther('1000'))  
  await vault.connect(user2).deposit(ethers.parseEther('2000'))
```

```
  expect(await  
    vault.totalSupply()).to.equal(ethers.parseEther('3000'))  
  })  
})
```

SECTION 9: SECURITY CHECKLIST

Before deploying any DeFi application:

Smart Contract Security

Re-entrancy protection on all external calls

Access control on privileged functions

- Integer overflow checks (or Solidity 0.8 +)
- Input validation on user-provided data
- Slippage protection on all swaps
- Deadline parameters on time-sensitive operations
- Emergency pause functionality
- Upgrade mechanisms (if needed) with timelock

Frontend Security

- Validate all user inputs before sending to chain
- Display clear warnings for high-risk actions
- Show gas estimates before confirmation
- Handle wallet disconnections gracefully
- Protect against phishing with clear domain indicators
- Validate contract addresses match expected values

Operational Security

- Multi-sig for contract ownership
- Timelock on governance actions
- Monitoring for unusual activity
- Incident response plan documented
- Bug bounty program active
- Regular security reviews scheduled

Testing Requirements

- 100% code coverage on critical paths
- Fuzzing on input validation
- Fork tests against live mainnet state
- Edge case testing (zero amounts, max amounts)
- Gas optimization verified

SECTION 10: COMPLETE APPLICATION EXAMPLE

Bringing everything together -- a complete yield aggregator

that integrates multiple concepts:

```
contract SimpleYieldAggregator {
    address public owner;
    address public guardian;

    bool public paused;

    address public depositToken;
    address public rewardToken;

    address[] public strategies;
    mapping(address => uint256) public strategyAllocations;
    mapping(address => bool) public approvedStrategies;

    mapping(address => uint256) public userDeposits;
    mapping(address => uint256) public userShares;

    uint256 public totalShares;
    uint256 public totalDeposited;

    uint256 public performanceFee;
    uint256 public managementFee;
    uint256 public constant FEE_DENOMINATOR = 10000;

    uint256 public lastHarvestTime;

    constructor(
        address deposit,
        address reward,
        uint256 perfFee,
        uint256 mgmtFee
    ) {
        owner = msg.sender;
        guardian = msg.sender;
        depositToken = deposit;
        rewardToken = reward;
    }
}
```

```
performanceFee = perfFee;  
managementFee = mgmtFee;  
lastHarvestTime = block.timestamp;  
}
```

```
modifier onlyOwner() {  
    require(msg.sender == owner);  
    _;  
}
```

```
modifier onlyGuardian() {  
    require(msg.sender == guardian || msg.sender == owner);  
    _;  
}
```

```
modifier whenNotPaused() {  
    require(!paused);  
    _;  
}
```

```
function deposit(uint256 amount) external whenNotPaused {  
    require(amount > 0);
```

```
    uint256 sharesBefore = totalShares;  
    uint256 depositsBefore = totalDeposited;
```

```
    IERC20(depositToken).transferFrom(msg.sender,  
    address(this), amount);
```

```
    uint256 newShares;  
    if (sharesBefore == 0) {  
        newShares = amount;  
    } else {  
        newShares = (amount * sharesBefore) / depositsBefore;  
    }
```

```
    userDeposits[msg.sender] += amount;
```

```
userShares[msg.sender] += newShares;
totalShares += newShares;
totalDeposited += amount;
```

```
_deployToStrategies(amount);
}
```

```
function withdraw(uint256 shares) external {
    require(shares > 0);
    require(userShares[msg.sender] >= shares);
```

```
    uint256 withdrawAmount = (shares * totalDeposited) /
    totalShares;
```

```
    userShares[msg.sender] -= shares;
    totalShares -= shares;
    totalDeposited -= withdrawAmount;
```

```
    _withdrawFromStrategies(withdrawAmount);
```

```
    IERC20(depositToken).transfer(msg.sender,
    withdrawAmount);
}
```

```
function harvest() external {
    uint256 totalHarvested = 0;
```

```
    for (uint256 i = 0; i < strategies.length; i++) {
        address strategy = strategies[i];
        uint256 harvested = IStrategy(strategy).harvest();
        totalHarvested += harvested;
    }
```

```
    if (totalHarvested > 0) {
        uint256 perfFeeAmount = (totalHarvested *
        performanceFee) / FEE_DENOMINATOR;
        IERC20(rewardToken).transfer(owner, perfFeeAmount);
```

```
uint256 remaining = totalHarvested - perfFeeAmount;  
totalDeposited += remaining;  
}
```

```
lastHarvestTime = block.timestamp;  
}
```

```
function addStrategy(address strategy) external onlyOwner {  
    require(!approvedStrategies[strategy]);
```

```
    approvedStrategies[strategy] = true;  
    strategies.push(strategy);  
}
```

```
function removeStrategy(address strategy) external  
onlyOwner {  
    require(approvedStrategies[strategy]);
```

```
    IStrategy(strategy).withdrawAll();  
    approvedStrategies[strategy] = false;
```

```
    for (uint256 i = 0; i < strategies.length; i++) {  
        if (strategies[i] == strategy) {  
            strategies[i] = strategies[strategies.length - 1];  
            strategies.pop();  
            break;  
        }  
    }  
}
```

```
function setAllocation(address strategy, uint256 allocation)  
external onlyOwner {  
    require(approvedStrategies[strategy]);  
    strategyAllocations[strategy] = allocation;  
}
```



```
function pause() external onlyGuardian {  
    paused = true;  
}
```

```
function unpause() external onlyOwner {  
    paused = false;  
}
```

```
function emergencyWithdraw() external onlyGuardian {  
    for (uint256 i = 0; i < strategies.length; i++) {  
        IStrategy(strategies[i]).withdrawAll();  
    }  
    paused = true;  
}
```

```
function _deployToStrategies(uint256 amount) internal {  
    uint256 totalAllocation = 0;  
    for (uint256 i = 0; i < strategies.length; i++) {  
        totalAllocation += strategyAllocations[strategies[i]];  
    }
```

```
    if (totalAllocation == 0) return;
```

```
    for (uint256 i = 0; i < strategies.length; i++) {  
        address strategy = strategies[i];  
        uint256 allocation = strategyAllocations[strategy];
```

```
        if (allocation > 0) {  
            uint256 deployAmount = (amount * allocation) /  
totalAllocation;  
            IERC20(depositToken).approve(strategy, deployAmount);  
            IStrategy(strategy).deposit(deployAmount);  
        }  
    }  
}
```

```
function _withdrawFromStrategies(uint256 amount) internal
```

```

{
    uint256 remaining = amount;
    uint256 available =
IERC20(depositToken).balanceOf(address(this));

    if (available >= amount) return;

    remaining -= available;

    for (uint256 i = 0; i < strategies.length && remaining > 0; i
+ +) {
        address strategy = strategies[i];
        uint256 strategyBalance = IStrategy(strategy).balance();

        if (strategyBalance > 0) {
            uint256 withdrawAmount = remaining > strategyBalance ?
strategyBalance : remaining;
            IStrategy(strategy).withdraw(withdrawAmount);
            remaining -= withdrawAmount;
        }
    }
}

function getShareValue() external view returns (uint256) {
    if (totalShares == 0) return 1e18;
    return (totalDeposited * 1e18) / totalShares;
}

function getUserValue(address user) external view returns
(uint256) {
    if (totalShares == 0) return 0;
    return (userShares[user] * totalDeposited) / totalShares;
}
}

interface IStrategy {
    function deposit(uint256 amount) external;

```

```
function withdraw(uint256 amount) external;  
function withdrawAll() external;  
function harvest() external returns (uint256);  
function balance() external view returns (uint256);  
}
```

This aggregator demonstrates production patterns:

- Multi-strategy allocation
- Share-based accounting
- Fee collection
- Emergency controls
- Guardian role for rapid response
- Pausable for emergencies

Building DeFi applications means accepting responsibility. Users trust you with real money. Every bug, every oversight, every unhandled edge case could cost someone their savings. Test thoroughly. Monitor constantly. Respond quickly. The decentralized financial system you're building changes lives -- make sure it's for the better.

This completes Book 2: DeFi and Tokenomics. You now understand how decentralized finance works from first principles to production code. The next step: apply this knowledge to build applications that serve users better than traditional finance ever could.